原创经典，程序员典藏

资深程序员10年开发经验的总结，深入剖析SQL Server 2012的精髓
全面涵盖SQL Server 2012数据库基础、安全、管理、开发及性能优化

SQL Server 2012
王者归来
——基础、安全、开发及性能优化

（40小时高清多媒体教学视频）

秦 婧 等编著



本书特色

- ◎ **内容全面**：涵盖了SQL Server 2012从入门到精通的方方面面内容
- ◎ **内容新颖**：紧跟数据库技术的最新趋势，总结了大量的全新观点和应用
- ◎ **示例丰富**：提供了670个示例，并用T-SQL语句和可视化操作两种方式实现
- ◎ **由浅入深**：从基本操作开始，逐步深入到数据库安全、开发和性能优化
- ◎ **讲解详细**：从概念、语法、示例、技巧和应用等多个角度进行分析
- ◎ **对比分析**：对SQL Server几个最常用版本的不同特性进行了对比分析
- ◎ **视频教学**：提供了40小时高清多媒体教学视频辅助学习，提高学习效率

超值、大容量DVD光盘

- ◎ 本书涉及的实例源文件
- ◎ 18小时高清配套教学视频
- ◎ 22小时SQL Server进阶视频
- ◎ 3部《程序员面试宝典》电子书

清华大学出版社

SQL Server 2012 王者归来

——基础、安全、开发及性能优化

秦 婧 等编著



清华大学出版社

北 京

内 容 简 介

本书由浅入深，全面细致地讲述了 SQL Server 2012 的功能特性和开发应用。从 SQL Server 数据库基础到数据库安全，再到 SQL Server 开发及数据库性能优化，涵盖 SQL Server 2012 的所有重要知识点。本书讲解时结合了大量实例，便于读者通过实践更加深刻地理解所学知识。**另外，作者专门为本书录制了 18 小时高清配套教学视频，与本书实例源文件一起收录于配书 DVD 光盘中。除此外，光盘中还赠送了 22 小时 SQL Server 学习视频和 3 部《程序员面试宝典》电子书，非常超值。**

本书共 22 章，分 4 篇。第 1 篇 SQL Server 基础，介绍 SQL Server 的发展历史、架构、安装及工具等，还介绍了 T-SQL 基础、数据库基本操作和 SQL Server 2012 的特色；第 2 篇数据安全，介绍 SQL Server 安全、数据文件安全与灾难恢复、复制；第 3 篇 SQL Server 开发，介绍数据库设计、SQL Server 与 CLR 集成、在 SQL Server 中使用 XML、使用 ADO.NET、使用 SMO 编程管理数据库对象、高级 T-SQL 语法、Service Broker——异步应用程序平台、空间数据类型、跨实例链接、数据库管理自动化及商务智能；第 4 篇数据库性能优化，介绍数据存储与索引、数据查询、事务处理和数据库系统调优工具。

本书内容全面，示例精巧而详尽，适合所有想全面学习 SQL Server 数据库技术的人员阅读，是各个层次的数据库学习人员和广大程序员学习 SQL Server 的极佳读物，更是 IT 开发人员的案头必备资料。

图书在版编目（CIP）数据

SQL Server 2012 王者归来——基础、安全、开发及性能优化 / 秦婧等编著. —北京：清华大学出版社，2014

ISBN 978-7-302-35518-2

Ⅰ. ①S…　Ⅱ. ①秦…　Ⅲ. ①关系数据库系统　Ⅳ. ①TP311.138

中国版本图书馆 CIP 数据核字（2014）第 032454 号

责任编辑： 夏兆彦
封面设计： 欧振旭
责任校对： 徐俊伟
责任印制：

出版发行： 清华大学出版社
　　网　　址： http://www.tup.com.cn, http://www.wqbook.com
　　地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
　　社 总 机： 010-62770175　　**邮　　购：** 010-62786544
　　投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
　　质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 48　　**字　　数：** 1195 千字
　　（附光盘 1 张）
版　　次： 2014 年 8 月第 1 版　　**印　　次：** 2014 年 8 月第 1 次印刷
印　　数： 1～ 000
定　　价： 　　元

产品编号：056101-01

前　　言

作为全新的企业级信息平台，SQL Server 2012 不仅提供了更高级别的企业级稳定性和更灵活而深入的商业智能，同时也提供了多种功能以满足公有云及私有云环境的应用实现与运行。所以很多人称 SQL Server 2012 为云端的数据库。本书的目的就是教会读者拨开云雾，看清 SQL Server 2012 的本质。

SQL Server 作为微软在数据库管理系统（DBMS）上的主打产品，经过了多个版本的改进后，在数据处理能力方面具备了良好的性能，从而占领了更加广阔的市场，成为世界三大数据库管理系统之一。作为开发人员和数据管理人员，不会 SQL Server 就相当于少了一项高级技能。

为了让更多的人更加系统、深入和透彻地学习 SQL Server，我们总结了多年的经验，编写了这本书。通过对本书内容的学习，读者无论对 SQL Server 数据库应用开发，还是对数据库管理，乃至对数据库性能调优等都会有新的认识和提高。可以说，本书是读者学习 SQL Server，了解其新特性，并扩展的 SQL Server 知识面的绝佳帮手。

本书讲解由浅入深，首先从最基础的数据库概念和 T-SQL 语法讲起，便于数据库初学者入门学习。接下来从数据库安全角度讲解了数据库内容的安全和数据文件的安全解决方案，从而加强读者在数据库应用开发中的安全意识。然后进入核心主题，从多个技术方向讲解了 SQL Server 2012 在开发中的特性和应用。最后进一步深入高级主题，讲解了 SQL Server 性能优化的相关知识。对于较难掌握的知识点，本书以精巧的示例来说明，清晰易懂。

本书涵盖的知识面广，对 SQL Server 2012 的大部分特性和功能都有所涉及。从最基础的 T-SQL 语法到 SQL Server 2012 中新增的高级语法，从数据库基本概念到新增的数据类型，从简单的数据库查询到数据库性能优化，从数据库的创建到数据库的管理，从数据库应用开发到商务智能应用等都有介绍。另外，编者还为本书内容录制了配套高清教学视频，以辅助读者更加高效、直观地学习，从而达到更好的学习效果。

本书特色

1. 观点新颖，紧跟趋势

在编写本书的过程中，编者查阅了大量国内外的最新技术文章，总结出了大量全新的观点和技术并应用到本书中，使得本书可以紧跟数据库技术趋势，适应技术的最新发展。

2. 内容丰富、涵盖广泛

本书所讲解的知识和内容主要针对 SQL Server 2012 版本，对于不推荐使用和过时的

语法及功能不作为重点进行讲解，并提示读者不要使用。本书涵盖的知识面广，在围绕 SQL Server 2012 讲解的基础上，还对 SQL Server 2000/2005/2008 几个版本中的不同特性进行了对比分析。

3．由浅入深，循序渐进

本书的编排采用循序渐进的方式，从最基本的数据库概念和数据库操作开始，逐步深入到数据库安全、开发和性能优化，适合读者系统地学习 SQL Server 2012 的使用、开发和管理。

4．示例丰富，实用性强

本书中使用了 500 多个规范的示例用于大部分知识点的演示和讲解，便于读者学习和理解。尤其是对于难度较高的知识点，本书使用精巧的示例化繁为简，便于读者掌握。这些示例简洁明了，读者可以按照示例进行实践和演练。书中重点总结了编者多年从事数据库管理和应用开发的经验，对于冷僻的问题基本不做过多介绍。

5．视频教学，高效直观

编者专门为本书录制了配套高清教学视频，便于让读者更加轻松、直观地学习，从而提高学习效率。这些视频与本书实例源文件一起收录于配书光盘中。

本书内容安排

第 1 篇　SQL Server 基础（第 1～4 章）

本篇讲述了 SQL Server 的基础知识，包括 SQL Server 的发展历史、SQL Server 2012 的安装和常用工具、T-SQL 语法、数据库对象等。概要性地介绍了 SQL Server 2012 的改进功能和新特性，为希望了解 SQL Server 数据库的新手提供一些基础知识。

第 2 篇　数据库安全（第 5～7 章）

本篇讲述了与 SQL Server 安全相关的各种知识，包括数据库加密、用户角色权限设置、数据库的备份与恢复、数据库快照、镜像、日志传送、数据库群集和数据库复制等。本篇通过详尽的操作步骤和多种示例让读者对数据库安全管理有一个初步的了解。

第 3 篇　SQL Server 开发（第 8～18 章）

本篇讲述了 SQL Server 在开发应用中的特性，主要包括数据库设计、CLR 集成、XML 的使用、ADO.NET 的使用、SMO 编程、高级 T-SQL 语法、Service Broker（即异步应用程序平台）、空间数据类型、跨实例链接、数据库管理自动化和商务智能等。本篇是本书的精华和核心所在，也是需要掌握的 SQL Server 2012 的核心知识。本篇所讲的内容是 SQL Server 2012 开发中的高级应用。通过对本篇内容的学习，读者可以了解和使用 SQL Server 2012 及其特性进行数据库应用开发。

第 4 篇　数据库性能优化（第 19～22 章）

本篇所介绍的数据库性能优化知识很容易被读者忽视。本篇内容也是 SQL Server 应用中最难掌握的知识。主要包括数据存储与索引、数据查询、事务处理、数据库系统调优工具等。本篇使用简单明了的示例来分析和介绍数据库性能优化，以小见大，帮助读者掌握数据库性能优化的知识。

超值 DVD 光盘内容

- ❑ 本书源程序；
- ❑ 18 小时高清配套教学视频；
- ❑ 10 小时 SQL Server 入门教学视频；
- ❑ 12 小时 SQL Server 进阶实例教学视频；
- ❑ 《C#与.NET 程序员面试宝典》电子书；
- ❑ 《C/C++程序员面试宝典》电子书；
- ❑ 《Java 程序员面试宝典》电子书；

本书读者对象

- ❑ SQL Server 入门新手；
- ❑ 想全面、系统、深入地学习 SQL Server 的人员；
- ❑ 想进一步提升 SQL Server 应用技能的人员；
- ❑ 具有 SQL Server 基础，想了解 SQL Server 2012 新特性的人员；
- ❑ 从事数据库应用开发，想对数据库管理和性能优化有所了解的开发人员；
- ❑ 从事.NET 应用开发，熟悉 C#语言的开发人员；
- ❑ 数据库技术爱好者和研究人员；
- ❑ 数据分析和设计人员；
- ❑ 大中专院校的学生；
- ❑ 社会培训班的学员；
- ❑ 需要一本案头必备手册的程序员。

本书作者

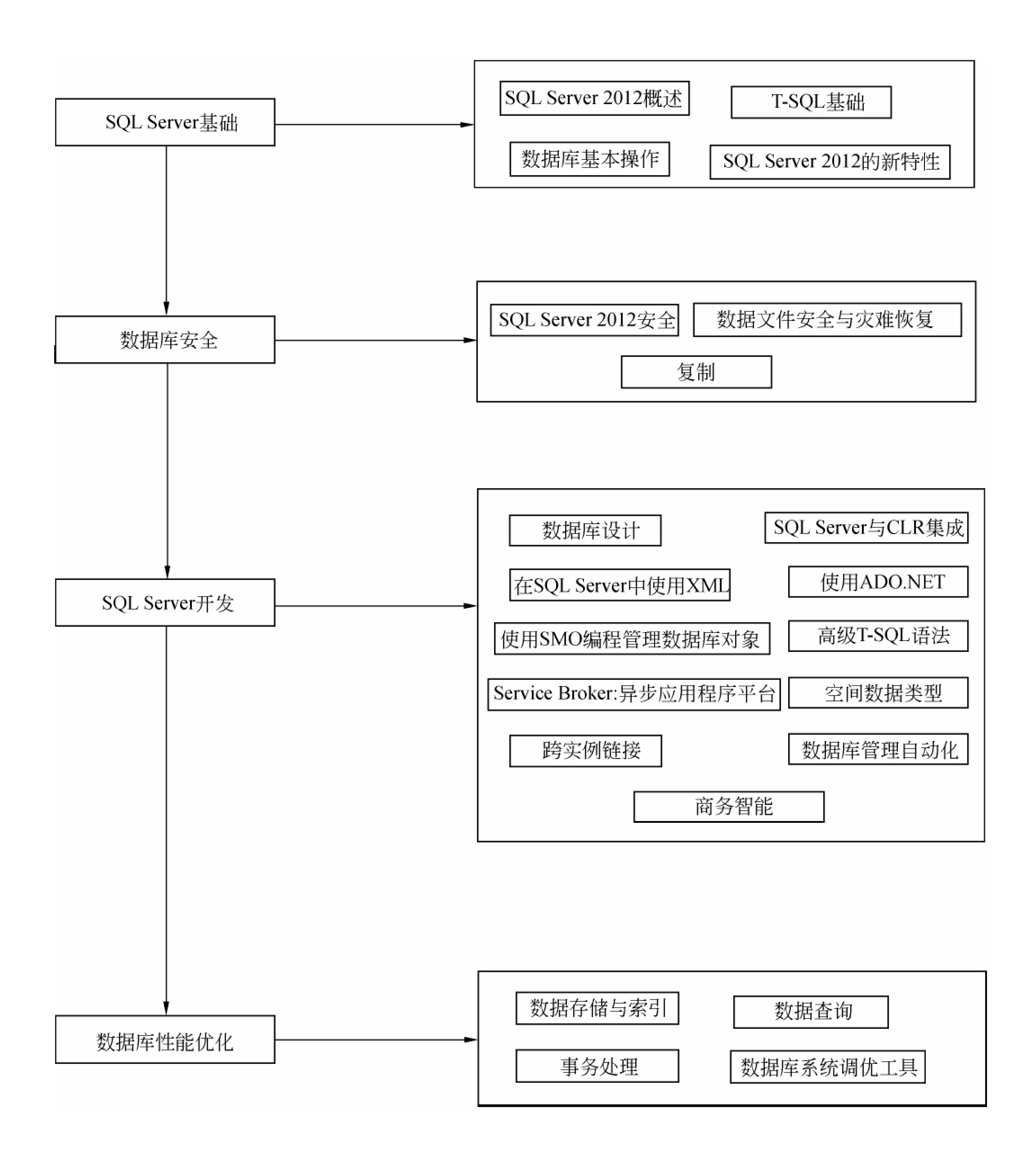
本书由秦婧主笔编写。其他参与编写的人员有陈小云、陈晓梅、陈欣波、陈智敏、崔杰、戴晟晖、邓福金、董改香、董加强、杜磊、杜友丽、范祥、方家娣、房健、付青、傅志辉、高德明、高雁翔、宫虎波、古超、桂颖、郭刚、郭立峰、郭秋滟、韩德、韩花、韩加国、韩静、韩伟、何海讯、衡友跃、李宁、李锡江、李晓峰、刘建准。

读者在阅读本时若有疑问，或者发现了本书中的不足和疏漏之处，请发电子邮件到 bookservice2008@163.com，编者会及时答复。

最后希望各位读者通过阅读本书，能很好地掌握 SQL Server 数据库技术，成为这个领域中的“王者”。笔者将倍感欣慰！所学授之于人，不亦乐乎？最后祝读书快乐！

编者

本书知识体系导读



SQL Server基础
SQL Server 2012概述
T-SQL基础
数据库基本操作
SQL Server 2012的新特性
数据库安全
SQL Server 2012安全
数据文件安全与灾难恢复
复制
SQL Server开发
数据库设计
SQL Server与CLR集成
在SQL Server中使用XML
使用ADO.NET
使用SMO编程管理数据库对象
高级T-SQL语法
Service Broker:异步应用程序平台
空间数据类型
跨实例链接
数据库管理自动化
商务智能
数据库性能优化
数据存储与索引
数据查询
事务处理
数据库系统调优工具

目　　录

第 1 篇　SQL Server 基础

第 2 篇 数据库安全

第 3 篇　SQL Server 开发

第 4 篇 数据库性能优化

第 1 篇　SQL Server 基础

第 1 章　SQL Server 2012 概述

SQL Server 作为一款面向企业级应用的关系数据库产品，在各行业和各软件产品中得到了广泛的应用，尤其是 SQL Server 2012 的发布使得 SQL Server 无论在效率上还是功能上较 SQL Server 2008 都有了很大的改善和提高。本章将主要讲解 SQL Server 2012 的基础知识及其安装和使用方法。

1.1　SQL Server 2012 简介

本节将主要介绍 SQL Server 的发展历史和特点，通过对数据库的发展历史和 SQL Server 的发展历史的了解，使读者更好地确定学习的目标。

1.1.1　SQL Server 发展历史

1946 年世界上第一台计算机 ENIAC 的诞生标志着人类进入了计算机时代。在使用计算机中必须面临的一个问题就是资料的存储。早期的计算机是将信息通过打孔的方式存储在纸带上，但是这种存储在纸带上的信息既不容易检索也不容易修改。后来随着磁存储介质的发明，信息才以文本文件或二进制文件的形式存储。这种以单独的文件来存放信息的方式就叫做文件处理系统（file-processing system）。

不同的信息被存放到不同的文件和不同的路径下，人们编写不同的应用程序来记录和处理需要的文件。文件处理系统的主要缺点如下：

- 无数据格式标准。由于文件和程序是在比较长的一段时间由不同的程序员编写的，而文件中并没有统一的格式来标注文件中的信息内容，容易造成对数据的理解不一致。比如一个学生管理系统，a 文件中记录了 a 学生选修的 5 门课程的成绩，b 文件记录的是 b 学生选修的 7 门课程的成绩。由于 a 和 b 选修的课程不同造成文件中的记录含义不同。由于没有统一的格式标准，大量的这种文件放在一起必然造成程序处理的困难。
- 数据冗余。采用文件存储的方式，由于缺乏唯一性检查，容易造成相同的信息在几个文件中重复存储。这种存储不但使得文件访问的开销增大，还会导致数据的不一致。
- 数据检索困难。由于文件系统中没有索引，若要检索出文件内容中的某行数据，程序就必须打开所有文件，找出其中符合条件的数据。还以前面提到的学生管理系统为例，若要找出英语分数最高的同学，程序必须打开每一个学生成绩的文件找到其中的英语成绩才能最终找到结果。对于几万或是几十万个文件，若要遍历

每一个文件，其处理效率可想而知。

传统的文件处理系统不支持以方便而高效的方式去获取所需数据。而随着计算机的普及，需要处理的数据不断膨胀，在面对几百万条、几千万条数据的情况下，文件处理系统已经无能为力。而且随着处理业务的不断复杂化，数据完整性问题、原子性问题、并发操作问题、数据安全问题等更使文件处理系统捉襟见肘。在这种情况下，数据库管理系统（DataBase Management System，DBMS）应运而生。

早期的数据库还是以数据存储和数据检索为主，使用网状数据模型和层次数据模型来描述数据、数据联系、数据定义和数据一致性约束。1970 年，美国 IBM 公司（主要产品为 DB2）的 E.F.Codd 在其发表的著名论文 *A Relational Model of Data for Large Shared Data Banks* 中首先提出了关系数据模型。后来 Codd 又提出了关系代数和关系演算的概念、函数依赖的概念、关系的三范式，为关系数据库系统奠定了理论基础。接着各大数据库厂商都推出了支持关系模型的数据库管理系统，标志着关系数据库系统新时代的来临。

随着关系数据库系统时代的到来，各大数据库厂商都开始推出自己的关系数据库产品。1989 年 Sybase 和 Ashton-Tate 公司（以其 dBase 软件成为当时数据库市场的霸主，1991 年被 Borland 并购）合作开发了数据库产品 SQL Server 1.0。而 Microsoft 为了能在关系数据库市场和甲骨文公司（主要产品 Oracle）以及 IBM 相抗衡，在 1992 年劝说 Sybase 公司进行 5 年的合作，共同研发数据库产品，并在之后推出了应用于 Windows NT 3.1 平台上的 Microsoft SQL Server 4.21 版本，这标志着 Microsoft SQL Server 的正式诞生。

20 世纪 90 年代，数据库市场百花齐放，竞争十分激烈。SQL Server 的早期版本由于其自身的不足，仅局限在小型企业和个人应用上。直到 1998 年 SQL Server 7.0 的推出才使 SQL Server 走向了企业级应用的道路。而随后发布的 SQL Server 2000 更是一款优秀的数据库产品，凭借其优秀的数据处理能力和简单易用的操作使得 SQL Server 跻身世界三大数据库之列（另外两个是 Oracle 和 IBM DB2）。表 1.1 给出了 SQL Server 各版本的发布时间和开发代号。

表 1.1　SQL Server各版本发布时间和开发代号

年　代	版　本	开 发 代 号
1993 年	SQL Server for Windows NT 4.21	无
1994 年	SQL Server for Windows NT 4.21a	无
1995 年	SQL Server 6.0	SQL 95
1996 年	SQL Server 6.5	Hydra
1998 年	SQL Server 7.0	Sphinx
2000 年	SQL Server 2000	Shiloh
2003 年	SQL Server 2000 Enterprise 64 位版	Liberty
2005 年	SQL Server 2005	Yukon
2008 年	SQL Server 2008	Katmai
2012 年	SQL Server 2012	Denali

虽然微软凭借 SQL Server 2000 成为世界数据库三巨头之一，但是与 Oracle 和 IBM 的 DB2 相比，SQL Server 2000 在数据处理效率、系统功能和市场占有率上仍有比较大的差距。

到 2004 年，据 IDC 统计，Oracle 的市场占有率为 41.3%，而 IBM 和微软的市场份额则分别为 30.6%和 13.4%。自从 2000 年微软发布 SQL Server 2000 以后，5 年来一直没有对 SQL Server 进行大的版本升级。

2005 年 SQL Server 2005 的发布可谓是微软在数据库市场投放的重磅炸弹，SQL Server 2005 不愧为微软“十年磨一剑”的精品之作。其高效的数据处理、强大的功能、简易而统一的界面操作，以及诱人的价格立即受到众多软件厂商和企业的青睐。SQL Server 的市场占有率不断增大，微软和 Oracle、IBM 又站在了同一起跑线上。

2008 年 SQL Server 2008 在原有 SQL Server 2005 的架构上做了进一步的更改。除了继承 SQL Server 2005 的优点以外，还提供了更多的新特性、新功能，使得 SQL Server 上升到新的高度。

2012 年 SQL Server 2012 在原有的 SQL Server 2008 的基础上又做了更大的改进。除了保留 SQL Server 2008 的风格外，还在管理、安全，以及多维数据分析、报表分析等方面有了进一步的提升。

1.1.2　SQL Server 2012 的特点

SQL Server 自从 6.0 版脱离 Sybase 架构后，每一个重大版本的发布都引入了新的特性和功能。

- ❑ SQL Server 7.0 使用了全新的关系引擎和查询引擎设计，并率先在数据库管理系统中引入 OLAP 和 ETL。这标志着 SQL Server 进入商务智能（BI）领域。
- ❑ SQL Server 2000 使得总体性能提高了 47%，同时增加了其扩展性和对 XML 的支持。另外 SQL Server 2000 还率先引入了通知服务、数据挖掘、报表服务等。
- ❑ SQL Server 2005 在性能上较 SQL Server 2000 有了更进一步的提高。在企业级数据管理平台方面的高可用性设计和全新的安全设计也特别引人注目。在商务智能数据分析平台上，SQL Server 2005 增强了 OLAP 分析引擎、企业级的 ETL 和数据挖掘能力。同时其还实现了与 Office 集成的报表工具。另外在数据应用开发平台上，SQL Server 2005 实现了与.NET 的集成、Web Service 集成、Native XML 支持以及 Service Broker 等。
- ❑ SQL Server 2008 除了在 SQL Server 2005 的基础上优化查询性能外，还提供了新的数据类型、支持地理空间数据库、增加 T-SQL 语法、改进了 ETL 和数据挖掘方面的能力。
- ❑ SQL Server 2012 在 SQL Server 2008 的基础上，新添加了 AlwaysOn 功能，提供了像 Oracle 数据库中的序列功能，以及新增 T-SQL 中的语法等内容。此外，还在商业智能方面提供了新的 PowerView 工具。

当然，作为微软在数据库市场的主打产品 SQL Server 2008 的升级版，SQL Server 2012 的特性不仅仅如此。微软官方网站给出了 SQL Server 2012 的关键功能列表，以供读者参考。

总体来说，SQL Server 正朝着更高的性能，更可靠和更安全的方向发展，并提供商务智能的集成，成为了集数据管理和分析于一体的企业级数据平台。

1.2　SQL Server 2012 架构简介

本节主要介绍 SQL Server 2012 的系统架构、数据存储方式、读写方式，以及 SQL 程序的运行方式。读者只有对系统架构有了一个基本的认识，才能更好地学习和理解 SQL Server 2012 的相关知识。

1.2.1　SQL Server 2012 系统架构

SQL Server 2012 功能模块众多，但是从总体来说可以将其分成两大模块：数据库模块和商务智能模块。

数据库模块除了数据库引擎以外，还包括以数据库引擎为核心的 Service Broker、复制、全文搜索等功能组件。而商务智能模块由集成服务（Integration Services）、分析服务（Analysis Services）和报表服务（Reporting Services）三大组件组成。各组件之间的关系如图 1.1 所示。

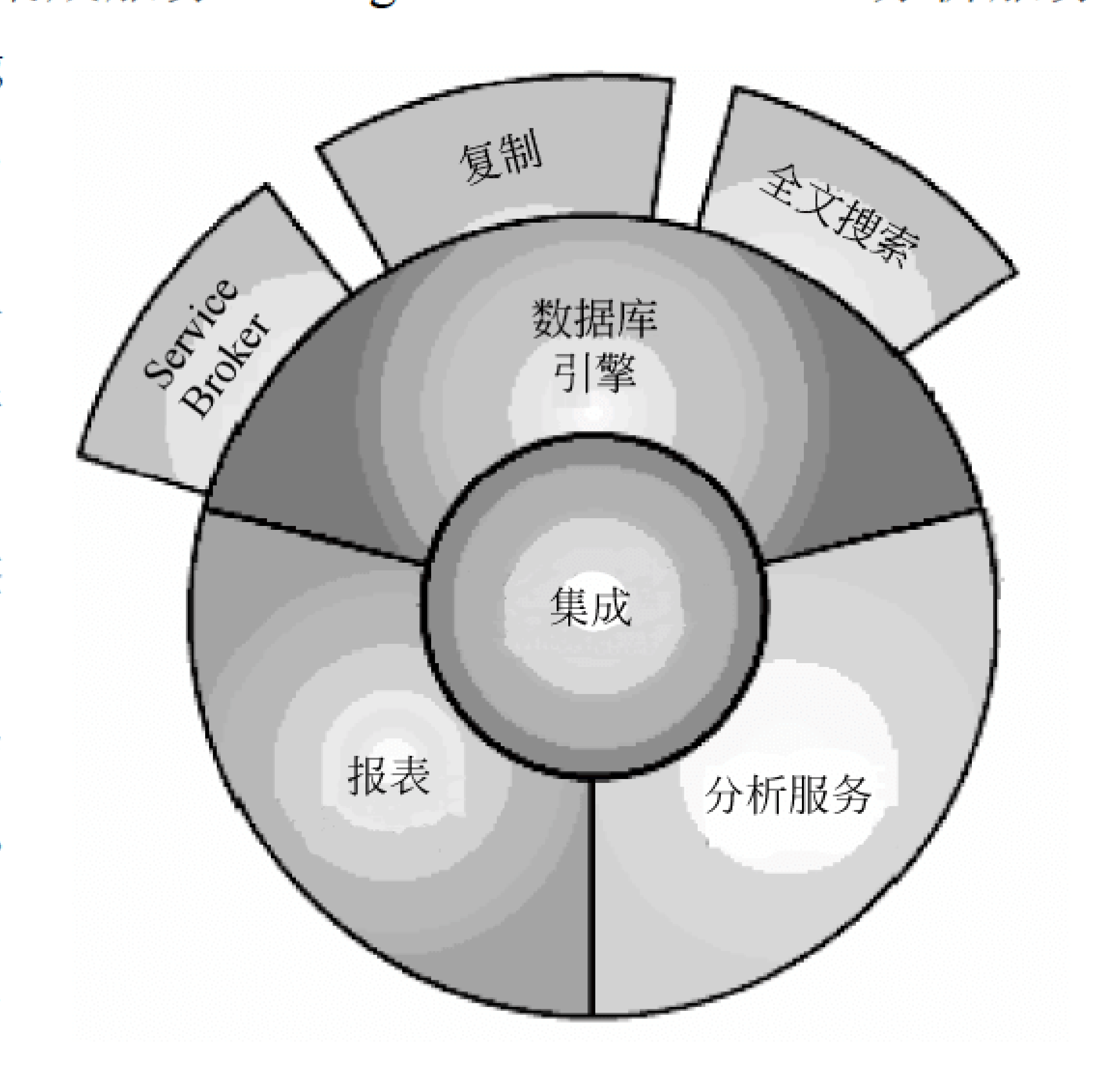


图 1.1　SQL Server 2012 组件之间的关系

从图 1.1 中可以看出，数据库引擎是整个 SQL Server 2012 的核心所在，其他所有组件都与其有着密不可分的联系。由于数据库引擎的重要性，这里主要讲解一下数据库引擎的内部架构。

图 1.2 显示了 SQL Server 2012 的总体结构。SQL Server 数据库引擎有 4 大组件：协议（Protocol）、关系引擎（Relational Engine，包括查询处理器，即 Query Compilation 和 Execution Engine）、存储引擎（Storage Engine）和 SQLOS。任何客户端提交的 SQL 命令都要和这 4 个组件进行交互。

协议层接受客户端发送的请求并将其转换为关系引擎能够识别的形式。同时，它也能将查询结果、状态信息和错误信息等从关系引擎中获取出来，然后将这些结果转换为客户端能够理解的形式返回给客户端。

关系引擎负责处理协议层传来的 SQL 命令，对 SQL 命令进行解析、编译和优化。如果关系引擎检测到 SQL 命令需要数据就会向存储引擎发送数据请求命令。

存储引擎在收到关系引擎的数据请求命令后负责数据的访问，包括事务、锁、文件和缓存的管理。

SQLOS 层则被认为是数据库内部的操作系统，它负责缓冲池和内存管理、线程管理、死锁检测、同步单元和计划调度等。

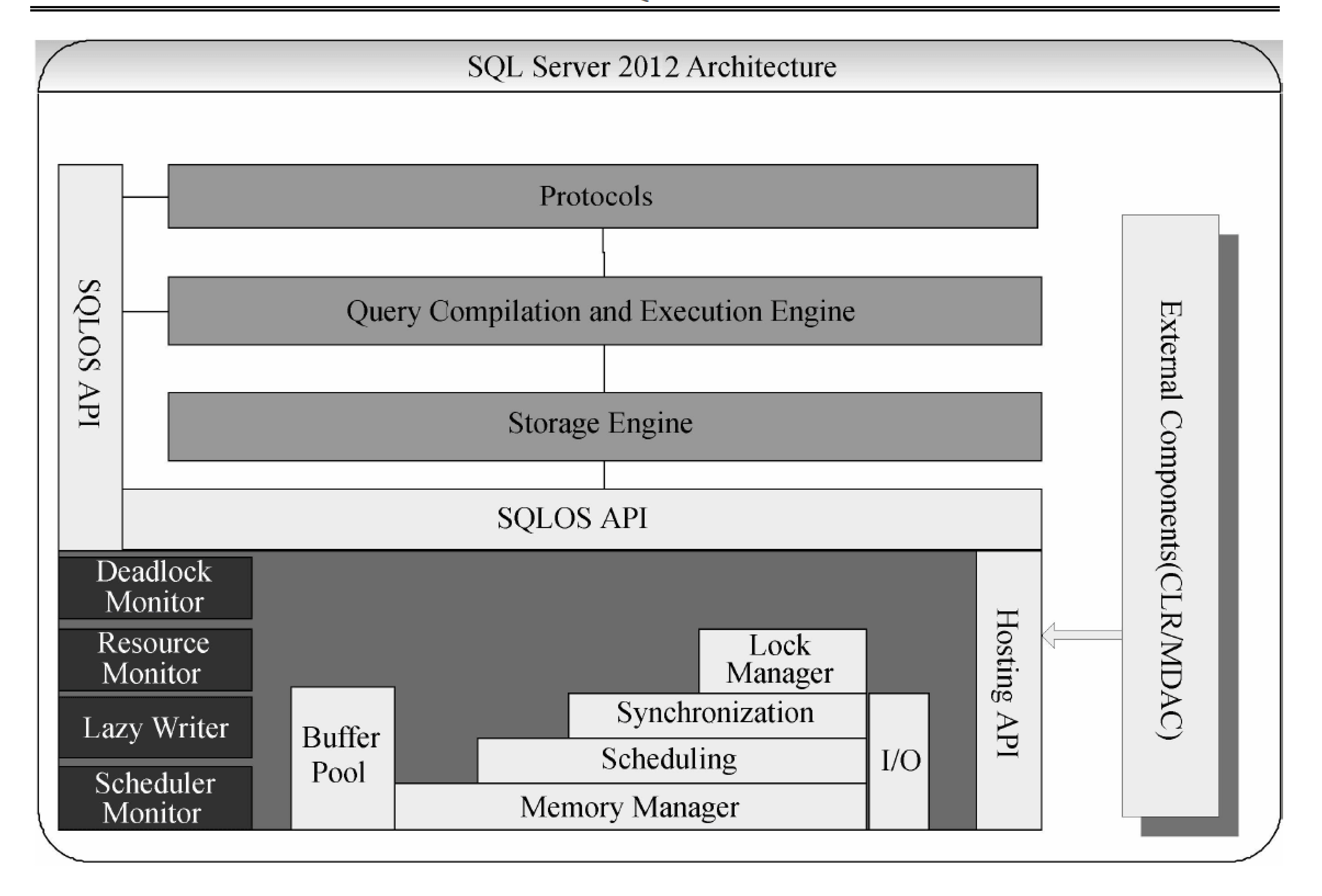


图 1.2　SQL Server 2012 架构

1.2.2　SQL Server 2012 的协议

当客户端向 SQL Server 发送 SQL 命令时，客户端发出的命令必须符合一定的通信格式规范才能被数据库系统识别，而这个规范就是 TDS（Tabular Data Stream）。服务器和客户端上都有 Net-Libraries，它可以将 TDS 信息包转换为标准的通信协议包。

SQL Server 可以同时支持来自不同客户端的多种标准协议，其支持的协议如下所述。

- 共享内存（Shared Memory）：这是 SQL Server 默认开启的一个协议。该协议简单，无需配置。顾名思义，共享内存协议就是通过客户端和服务端共享内存的方式来进行通信。所以使用该协议的客户端必须和服务端在同一台机器上。由于共享内存协议简单，协议效率高而且安全，所以如果客户端（比如 IIS）和数据库是在同一台机器上，那么使用共享内存协议是一个不错的选择。
- 命名管道（Named Pipes）：该协议是为局域网而开发的协议。命名管道协议和 Linux 下的管道符号有点接近，一个进程使用一部分内存来向另一个进程传递信息，一个进程的输出是另一个进程的输入。两个进程可以是同一台机器，也可以是局域网中的两台机器。
- TCP/IP：该协议是因特网上广为使用的协议。该协议可以用于不同硬件、不同操作系统、不同地域的计算机之间通信。由于 TCP/IP 协议没有共享内存协议和命名管道协议的限制，所以该协议在 SQL Server 上被大量使用。

1.2.3　SQL Server 2012 的查询

查询处理器由解析器、优化器、SQL 管理器、数据库管理器和查询执行器组成。它主要负责 SQL 命令处理。查询处理器是整个 SQL Server 中最为复杂的组件，其性能的好坏就决定了整个 SQL Server 数据处理能力的高低。当一个 SQL 命令从协议层传输到查询处理器时，各模块的分工如下所述。

- 命令解析器首先接收到协议层传来的 T-SQL 语句。命令解析器首先对 T-SQL 语法进行检查。如果解析器无法正确识别语法，则直接抛出错误并标出错误的地方。在语法检查通过后命令解析器会将 SQL 命令翻译成查询树，并将查询树传给查询优化器。至此命令解析器的任务结束，而源 SQL 命令也将不再可用。
- 查询优化器负责查询树的执行优化并生成最终的执行计划。查询优化器从命令解析器中获得查询树后，将不能优化的控制流 DDL 命令等编译成一种内部格式，而可以优化的 DML 语句（如 select、insert、update 和 delete）将由查询优化器进一步判断最佳的处理方式。对于可优化语句，查询优化器先将每个查询进行规范化，然后基于成本选择成本最低的执行计划。执行成本以内存使用量、CPU 使用率和 I/O 数量为依据。查询优化器会考虑语句的类型并检查受影响的各个表的数据量，查询每张表中可用的索引和统计信息来决定最优的执行计划。在规范化和优化完成后，查询树会被编译成执行计划。执行计划实际上是一种数据结构，其中包含了每个命令将会影响的表，会使用的索引，进行安全检查和必须判断为真的选择条件。
- SQL 管理器负责管理与存储过程（Stored Procedure，SP）执行计划有关的一切事务。SQL 管理器会判断什么时候一个执行计划需要重新编译并管理存储过程缓存以便其他进程重用这些缓存。另外，SQL 管理器还负责管理查询的参数自动化，也就是说 SQL 管理器可以从某些 SQL 命令中提取出参数，而将参数形式的 SQL 命令的执行计划缓存起来，从而提高 SQL 语句查询的效率。
- 数据库管理器管理查询编译和查询优化所需的对元数据的访问。
- 查询执行器运行查询优化器生成的执行计划。该模块逐步运行执行计划中的每一个命令，管理其中的事务和锁，并将需要数据操作的执行计划传入存储引擎。

如图 1.3 所示为 SQL Server 2012 的查询流程图。

1.2.4　SQL Server 2012 的数据操作

SQL Server 2012 的数据操作主要由存储引擎来完成。当查询处理器向存储引擎发出数据操作请求时，存储引擎会调用存取方法的代码向缓存管理器发出请求，缓存管理器负责从缓存中提供数据或者从硬盘上把数据读取到缓存中，下次再查询该数据时只要查询处理器就可以直接从缓存中读取数据，而不需要进行硬盘的 I/O 操作。

如图 1.4 形象地表示了数据的读取方式。由于内存的访问速度远远高于硬盘的访问速度，这种缓存读取的方式尽量减少了硬盘的读写操作，从而大大提高了数据处理效率。

SQL Server 在数据访问中的最小单位是页（Page）。也就是说，即使只需要查询一个字节的数据，SQL Server 至少也要访问一个页来查找结果。每个数据库都是由页组成的集合。一个页的大小是 8KB，而 8 个连续的页组成了一个区（Extent）。

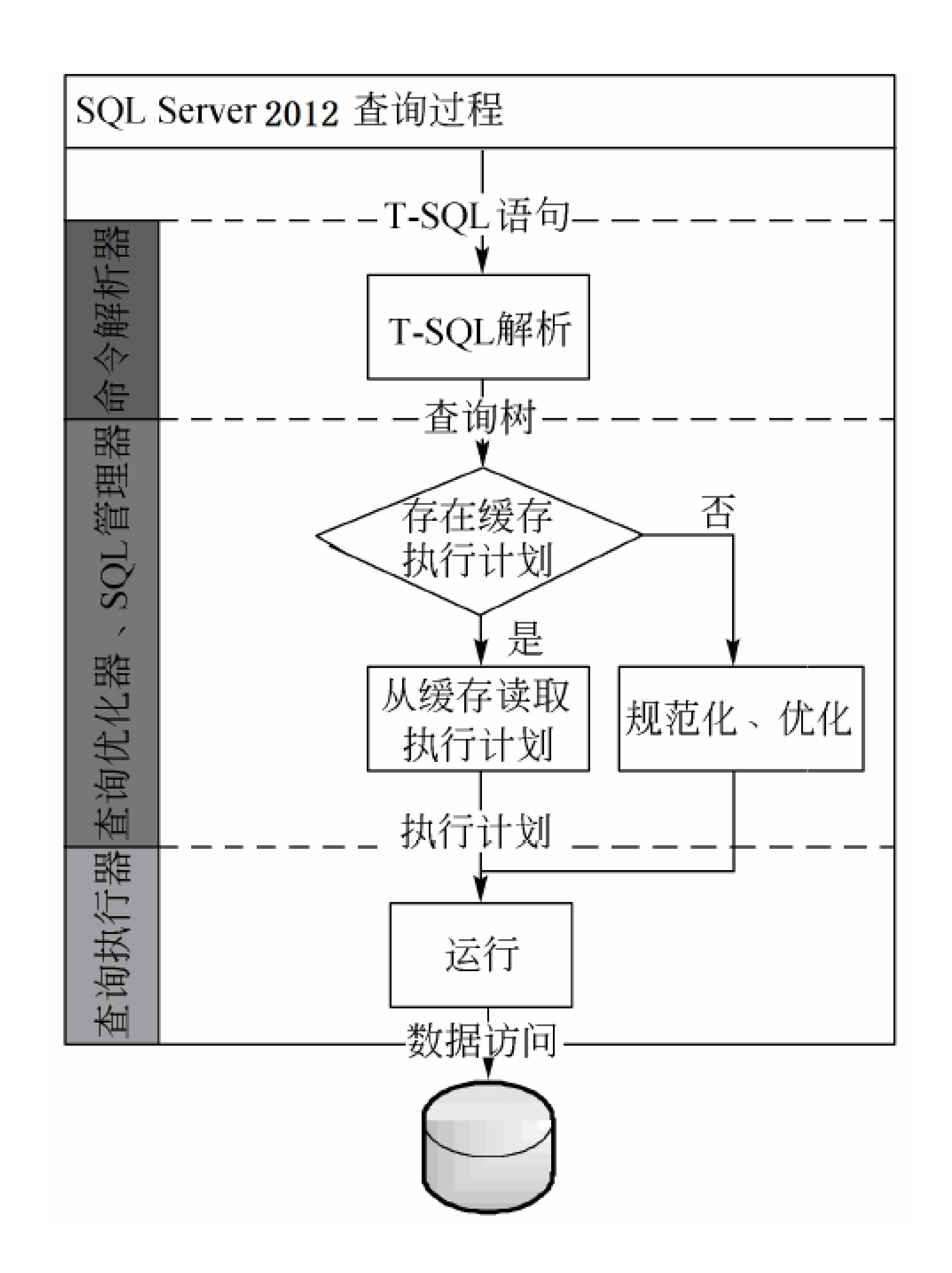


图 1.3　查询过程

SQL Server 中使用了 8 种类型的页：数据页、LOB（大数据类型）页、索引页、页面自由空间页（PFS）、全局分配图和共享全局分配图页（GAM 和 SGAM）、索引分配图（IAM）页、大批量修改图（BCM）页和增量修改图（DCM）页。

虽然每个页的大小是 8KB，但是 SQL Server 中规定表中行（不包括可变长度数据类型的列）的最大长度是 8060 字节。所有的用户数据都保存在数据页或大型数据页上，所有的索引行都保存在索引页上。PFS 页用来跟踪数据库中哪些页是空的，可以用来记录新数据。分配页（GAM、SGAM 和 IAM）用来跟踪其他页面，不含任何用户数据并且只能在内部使用。大批量修改图页和增量修改图页用来使数据库备份和恢复效率更高。

图 1.4 展示的是数据的读取方式，但是对于数据的写入，SQL Server 的处理方式有一定的不同。为了保证事务的原子性——一个事务要么全部做完，要么什么都不做，但数据库必须要实现事务的回滚。如果一个事务要先修改 A 表然后修改 B 表，当事务修改完成 A 表正要修改 B 表时系统发生了故障，那么该事务对 A 表的修改必须回滚到事务还没有开始前 A 表的状态。先写日志技术使得事务回滚成为可能。先写日志技术如图 1.5 所示，SQL Server 是先将数据修改操作在日志文件中进行，直到事务提交时才将对日志中的修改同步到数据文件中。若事务被回滚，只需要恢复日志文件中的修改，数据文件并未修改过。

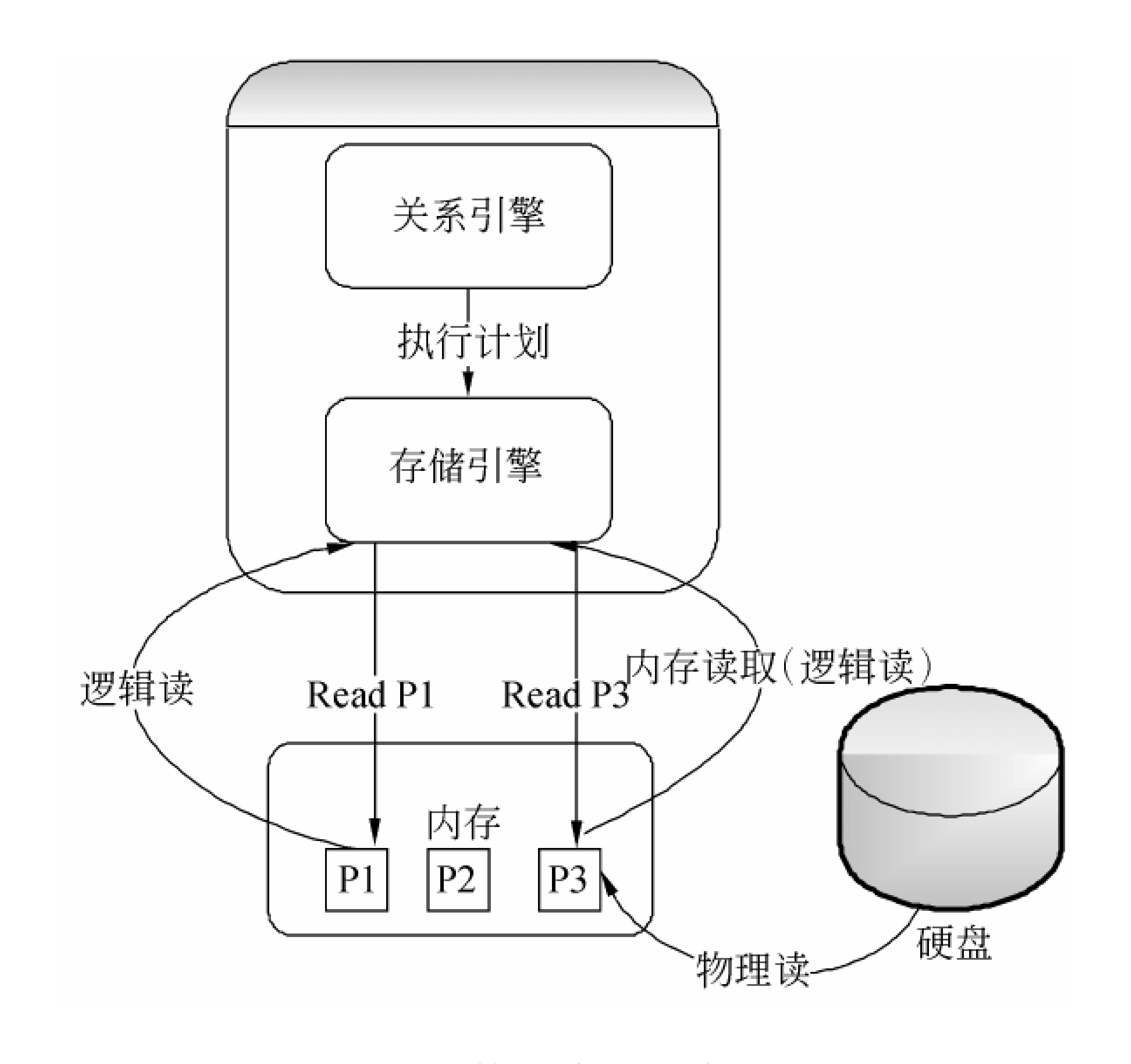


图 1.4　数据读取方式

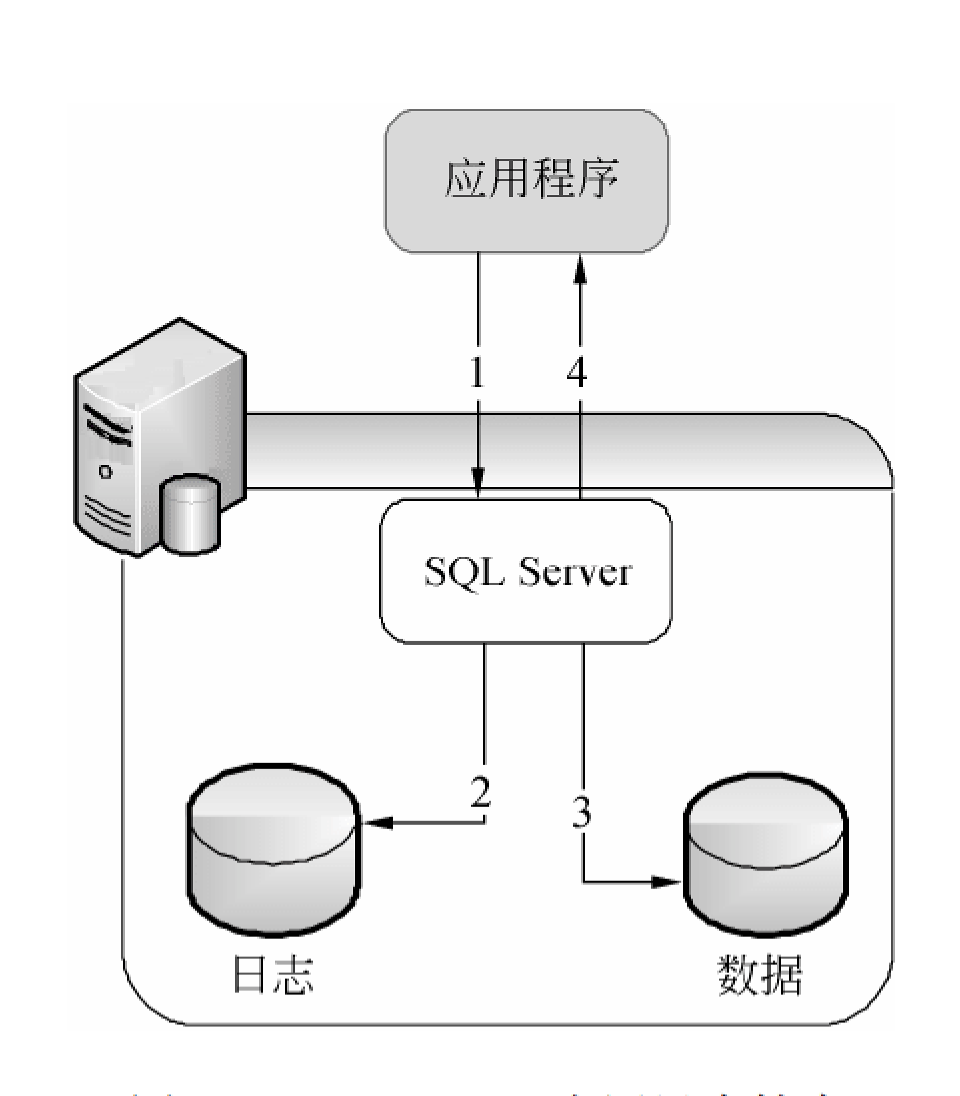


图 1.5　SQL Server 先写日志技术

另外，为了提高 SQL Server 的性能，采用了“懒写入”（Lazy Write）技术。即对日志文件的修改和数据文件的修改都是在内存中的修改，SQL Server 并不会立刻把修改写入硬盘。懒写入技术使得对未写入硬盘的数据页的回滚成为可能。

除了一般的数据查询和数据写入操作外，存储引擎还要负责事务隔离级别的控制、锁的控制、批量装载、DBCC 命令、备份和恢复操作等功能。这些功能将在接下来的章节进行详细的介绍。

1.3　SQL Server 2012 的安装

通过前两节的学习，相信读者对 SQL Server 已经有了一个初步的认识。本节将主要讲解 SQL Server 2012 的安装，为以后 SQL Server 的使用做环境准备，并正式开始踏上 SQL Server 2012 的学习之旅。

1.3.1　SQL Server 2012 的版本选择

根据数据库应用环境的不同，SQL Server 2012 发行了不同的版本以满足不同的需求。总的来说，SQL Server 2012 主要包括 4 种主要版本：精简版（SQL Server 2012 Express Edition）、商业智能版（SQL Server 2012 Bussiness Intelligence Edition）、标准版（SQL Server 2012 Standard Edition）和企业版（SQL Server 2012 Enterprise Edition）。每个版本的主要特点如下所述。

1．精简版

免费的精简版与其前身 MSDE 相似，使用核心 SQL Server 数据库引擎。但其缺少管理工具、高级服务（如 Analysis Services）及可用性功能（如故障转移）。

然而，精简版在一些关键方面对其前身进行了改进。其中最值得一提的是，微软消除了 MSDE 的“节流”限制——在数据库同时处理超过 5 个查询时性能下降。

精简版限于不超过 1GB 的内存，而且只能使用单颗处理器运行（而在 MSDE 可以访问两颗处理器和 2GB 内存）。

精简版的每个实例可支持高达 4GB 的数据库，而 MSDE 是 2GB 的限制。

精简版包含 Reporting Services。此版本仅能使用 SQL Server 关系数据库作为报表数据源，并且那些数据库必须位于运行报表服务器的物理机器上。

此外，精简版不包含 Report Builder 功能。

说明：精简版是完全免费的。若用户需要使用精简版 SQL Server 可以到微软官方网站下载。

2．商业智能版

SQL Server 2012 的商业智能版主要是应对目前数据挖掘和多维数据分析的需求应运

而生的。它可以为用户提供全面的商业智能解决方案，并增强了其在数据浏览、数据分析和数据部署安全等方面的功能。

3．标准版

标准版对与之对应的 SQL Server 2012 标准版进行了更新，保持四颗处理器的限制，但消除了 2GB 内存的上限。有两种针对 Itanium 和 X86 X64 处理器的版本，允许服务器访问大量内存。

标准版包含 Integration Services，带有企业版中可用的数据转换功能的子集。例如，标准版包含诸如基本字符串操作功能的数据转换，但不包含数据挖掘功能。标准版还包括 Analysis Services 和 Reporting Services，但不具有在企业版中可用的高级可伸缩性和性能特性。

标准版中的 Reporting Services 可以使用关系及非关系数据源（如 OLAP 多维数据集），并可以使用不同 SQL Server 的数据库系统。

4．企业版

企业版位于产品系列的高端，消除了大部分可伸缩性限制。其支持任意数量的处理器、任意数据库尺寸，以及数据库分区。

企业版包含所有 BI 平台组件功能齐备的版本。Integration Services 包含所有的数据转换功能。企业版中的 Analysis Services 获得改进的性能和可伸缩性功能，如主动缓存、跨多个服务器对大型多维数据库进行分区的功能。

与标准版相同，企业版中的 Reporting Services 可以使用关系及非关系数据源，并可以使用不同于 SQL Server 的数据库系统。它还得到高级可伸缩性功能，管理员可以配置 Reporting Services 群集。其中，多个报表服务器共享单个报表服务器数据库。如表 1.2 列出了各版本的 SQL Server 2012 之间的差异，以方便读者查看。

表 1.2　各版本的SQL Server 2012 比较

	精　简　版	商 业 智 能 版	标　准　版	企　业　版
使用的最大空间（每一个数据库实例）	1GB	64GB	64GB	操作系统支持的最大值
集成服务	仅有数据的导入和导出、内置数据源连接器	支持基本功能	支持基本功能	支持全部功能
分析服务	无	支持	不支持可扩展的共享数据库（附加/分离，只读数据库），其他的都支持	支持
报表服务	不支持	支持		支持

另外，微软还发布了开发者版（SQL Server 2012 Developer Edition）和 180 天评估版（SQL Server 2012 Evaluation Edition）等。但是这些版本由于许可证限制，一般不用于生产

服务器，所以在此不做比较。

除了使用在 PC 和服务器上的版本外，SQL Server 2012 还有一个移动版（Compact Edition）。移动版是一个免费的嵌入式 SQL Server 数据库，可以用于创建移动设备、桌面端和 Web 端独立运行的和偶尔连接的应用程序。

说明：开发者版和评估版都包含企业的所有功能，若读者希望使用 SQL Server 的所有功能而没有企业版，那么可以使用开发者版和评估版。

1.3.2　SQL Server 2012 的安装环境

SQL Server 2012 各版本除了在 CPU 个数、内存使用量、数据库容量和功能模块等方面有限制外，还对操作系统、CPU 类型、应用软件等有不同的要求。

- 精简版 SQL Server 提供了 32 位和 64 位的版本，它可以运行在 Windows 7、Windows 8、Windows Server 2008、Windows Server 2012 和 Vista 等操作系统上。
- 商业智能版提供了 32 位和 64 位的版本，它只能运行在 Windows Server 2008、Windows Server 2012 的版本操作系统上。
- 标准版同时提供了 32 位和 64 位版。它可以运行在 Windows 7、Windows 8、Windows Server 2008、Windows Server 2012 和 Vista 等操作系统上。
- 企业版同商业智能版相同，提供了 32 位和 64 位版本，而且只能运行在 Server 版的操作系统上。

另外，Reporting Service 是发布在 IIS 上的，所以安装 Reporting Service 时必须先在操作系统中安装 IIS。其他一些支持文件如.NET Framework，则会在安装 SQL Server 2012 的同时自动安装到系统中。

1.3.3　安装配置 SQL Server 2012

在获得了需要安装的 SQL Server 光盘或安装文件，并确认计算机的操作系统、硬件和相关软件满足该版本的 SQL Server 的需求后，就可以安装配置 SQL Server 2012 了。

技巧：在 XP 和 Vista 操作系统下无法安装 SQL Server 2012 企业版。若读者希望安装 SQL Server 2012 企业版用于学习而且有较大的内存，那就不必在计算机中重新安装 Windows 2008 的操作系统。读者可以使用虚拟机 Virtual PC（Home 版无法安装）将 Windows 2008 安装到虚拟机中，然后在虚拟机中就可以安装 SQL Server 2012 企业版了。

SQL Server 2012 的具体安装步骤如下所述。

（1）将 SQL Server 的安装光盘放入光驱。若使用镜像文件安装则使用虚拟光驱工具将镜像文件载入虚拟光驱。

（2）双击光盘驱动器，安装程序将检测当前的系统环境。如果没有安装.NET Framework 3.5 SP1，将先安装该软件。

（3）安装程序检测当前系统的补丁。如果必需的系统补丁并未安装，则会安装系统

补丁。

（4）安装补丁后重启系统。再次双击光盘驱动器，SQL Server 2012 安装中心将启动。单击“安装”选项，切换到安装界面，如图 1.6 所示。

（5）单击“全新 SQL Server 独立安装或向现有安装添加功能”选项，系统将打开 SQL Server 2012 的安装程序，并检测当前环境是否符合 SQL Server 2012 的安装条件，如图 1.7 所示。

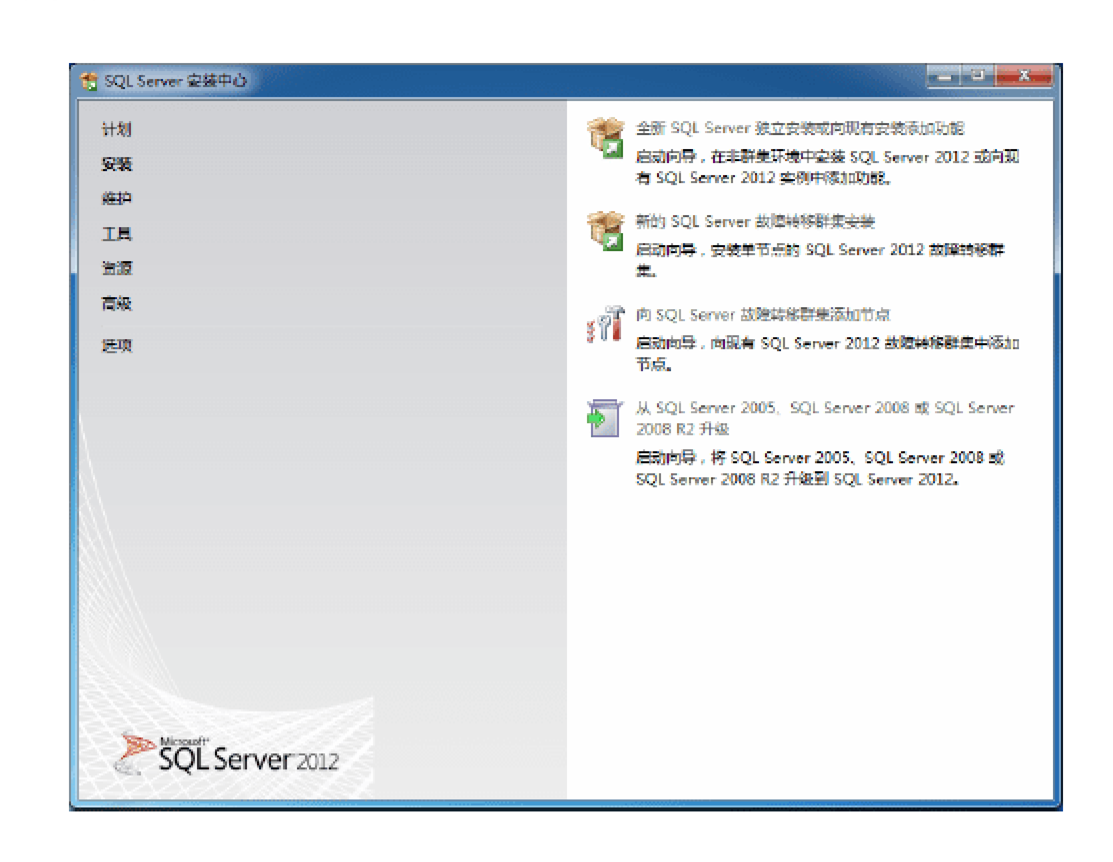


图 1.6　SQL Server 2012 安装中心

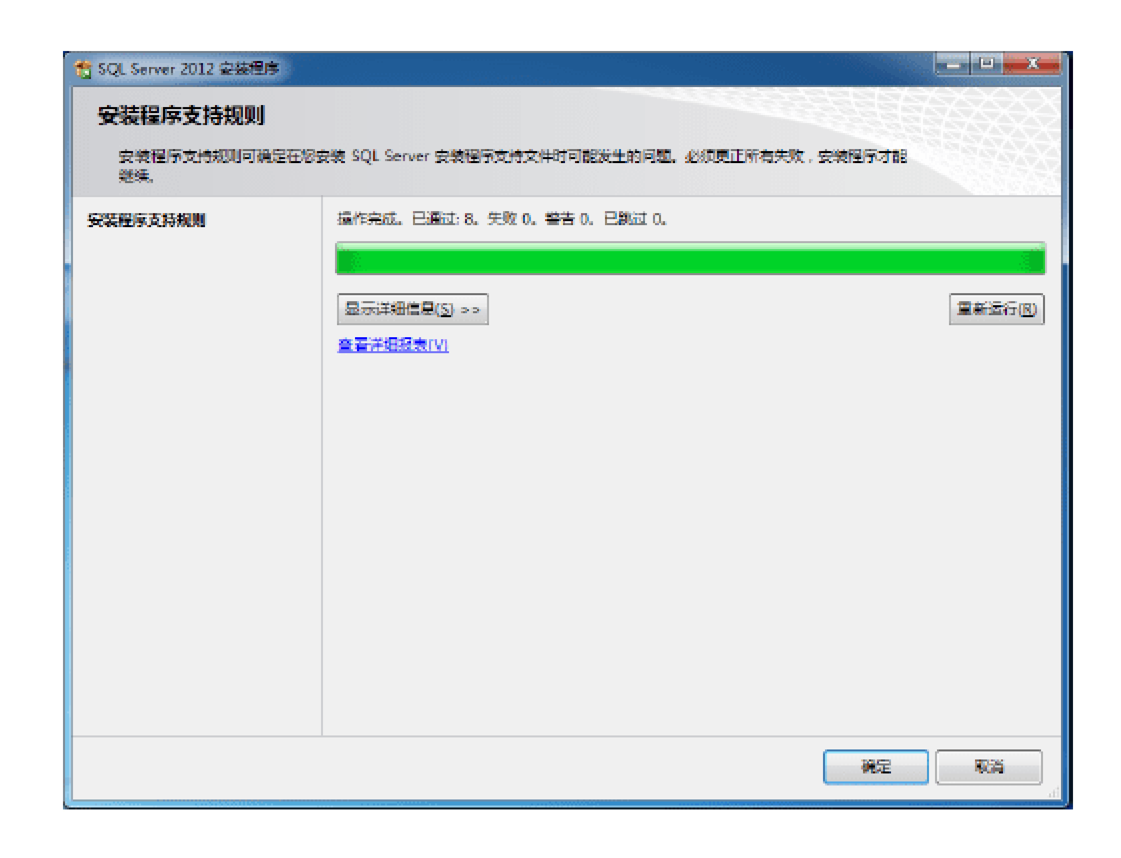


图 1.7　SQL Server 2012 安装程序界面

（6）单击“确定”按钮，进入产品密钥设置界面。输入产品密钥，然后接受许可条款。单击“安装”按钮，系统将安装程序支持文件。安装完支持文件后，系统将再次检测安装程序支持规则，如图 1.8 所示。

（7）单击“下一步”按钮，进入功能选择界面，如图 1.9 所示。

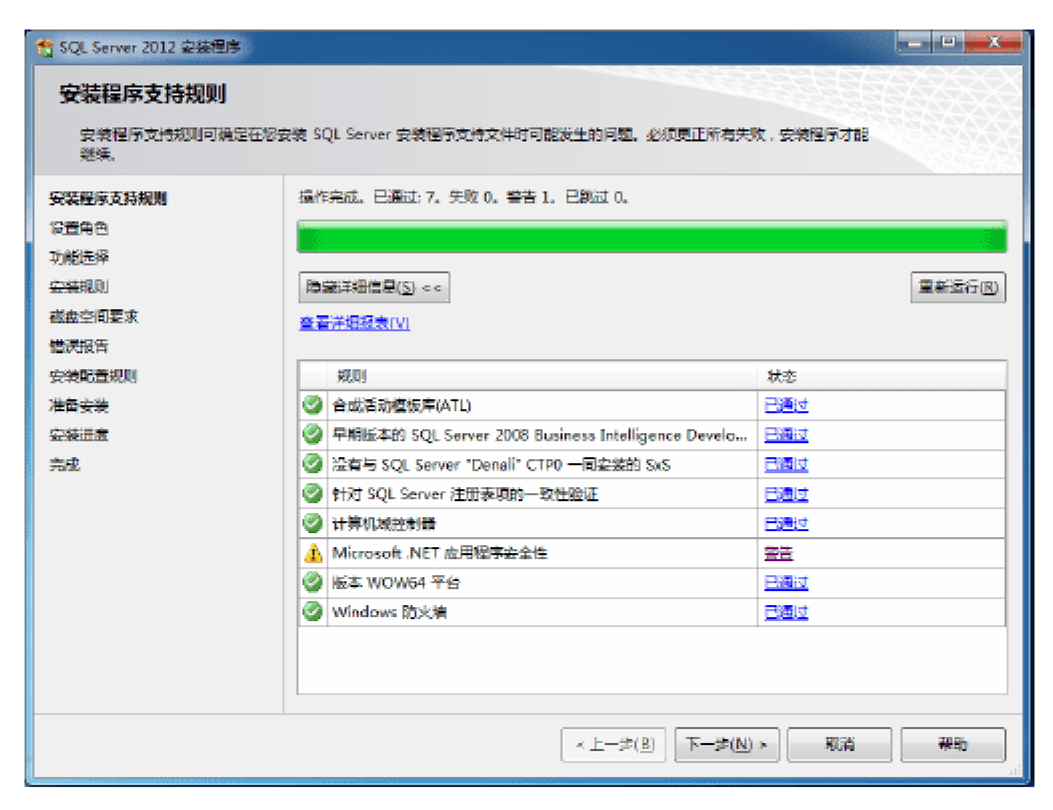


图 1.8　检测安装程序支持规则

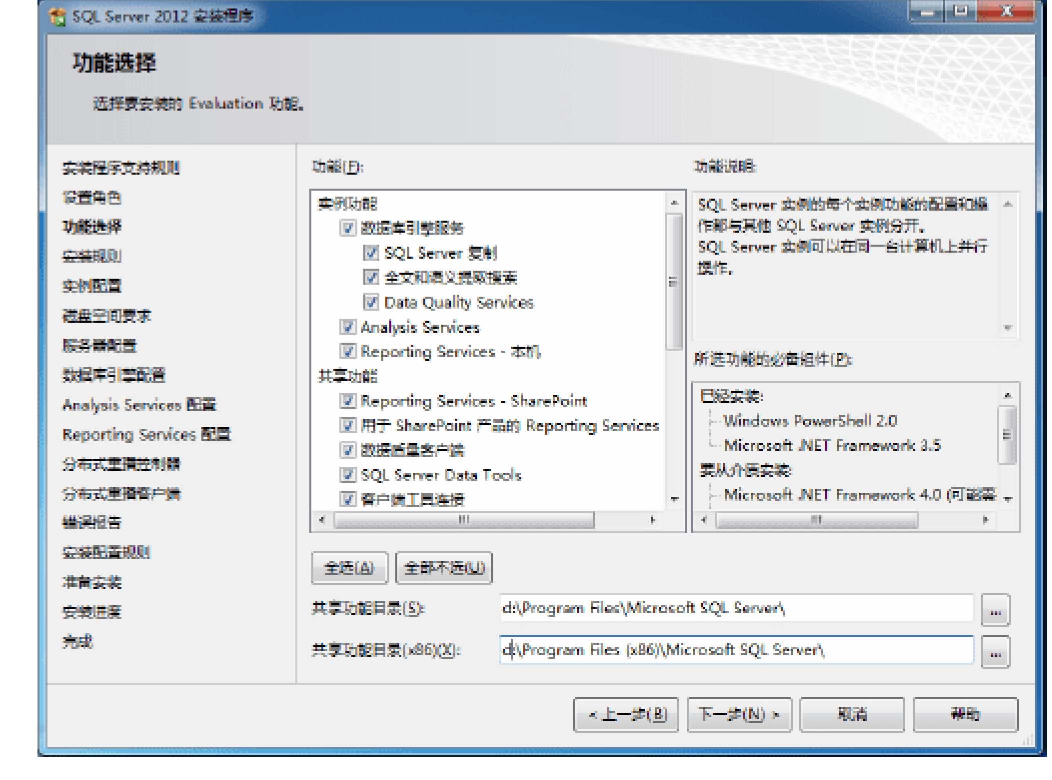


图 1.9　功能选择

这里将根据实际需要来选择安装对应的功能模块，如果出于学习的目的而不是安装到正式环境中，则可安装所有的功能模块。另外该界面还可以修改安装目录。

（8）单击“下一步”按钮，进入实例配置界面，如图 1.10 所示。

如果需要安装成默认实例，则选择“默认实例”单选按钮，否则选择“命名实例”单选按钮并在文本框中输入具体的实例名。SQL Server 允许在同一台计算机上同时运行多个实例。这里安装默认实例，其他选项采用默认值即可。

（9）单击“下一步”按钮，进入磁盘空间要求界面。该界面列出了安装 SQL Server 2008

需要的硬盘空间大小。

（10）单击“下一步”按钮，进入服务器配置界面。该界面主要配置服务的账户、启动类型、排序规则等，如图 1.11 所示。

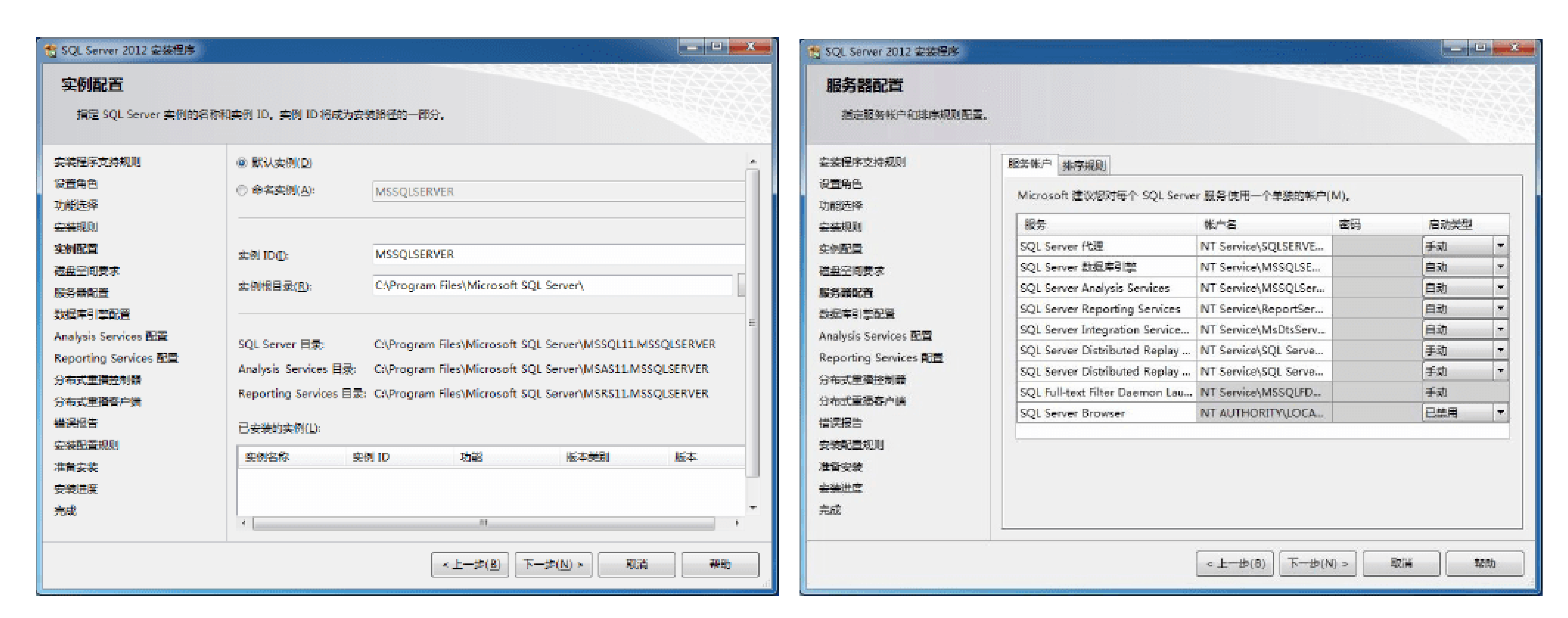


图 1.10　实例配置界面　　　　图 1.11　服务器配置界面

这里将账户名设置为 SYSTEM。由于 SQL Server Analysis Services 和另外两个服务是商务智能中使用的，一般情况下不使用，所以将其启动类型设置为手动。SQL Server 代理设置为手动，在需要使用的时候启动。排序规则一般情况下采用默认值即可。

注意：如果账户名设置错误，系统将会提示，而且也不能执行下一步操作，所以必须确保每个服务的账户名都正确。

（11）单击“下一步”按钮，进入数据库引擎配置界面，用于配置数据库账户、数据目录和 FILESTREAM，如图 1.12 所示。

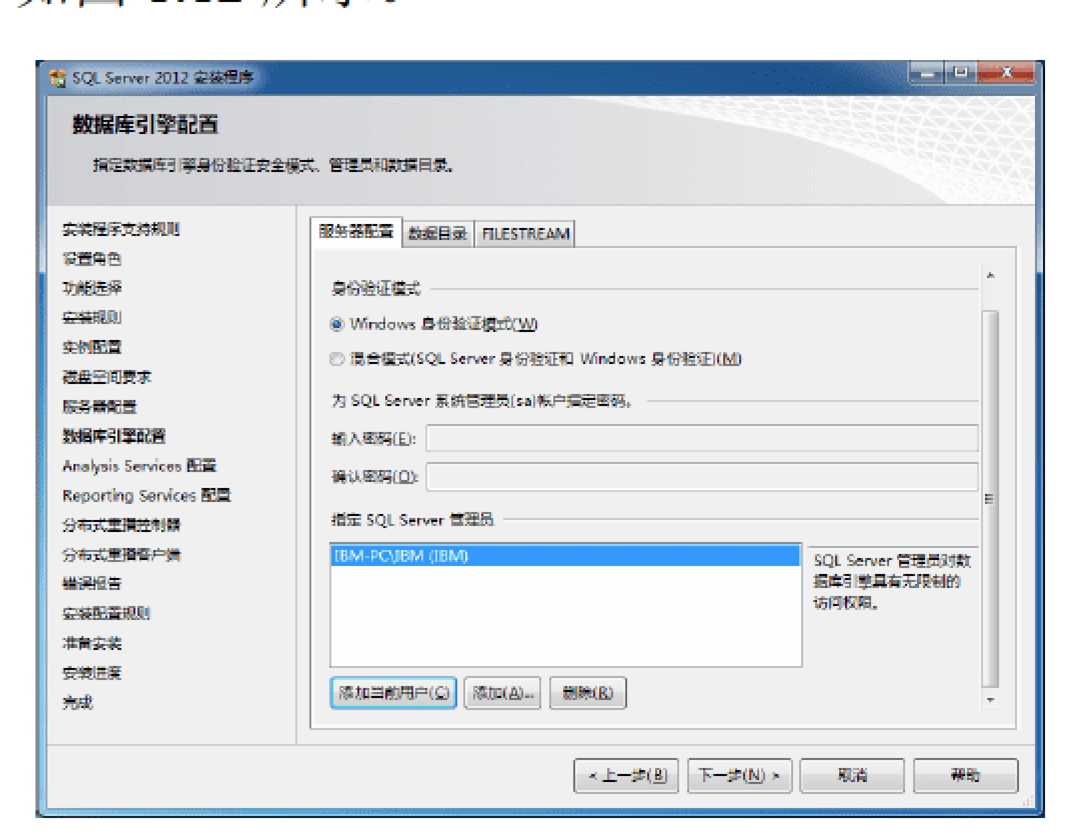


图 1.12　数据库引擎配置界面

在 SQL Server 2012 中有两种身份验证模式：Windows 身份验证模式和混合身份验证模式。Windows 身份验证模式是只允许 Windows 中的账户和域账户访问数据库；而混合身份验证模式除了允许 Windows 账户和域账户访问数据库外，还可以使用在 SQL Server 中配置的用户名密码来访问数据库。

如果使用混合模式则可以通过 sa 账户登录，在该界面中则需要设置 sa 的密码。单击“添加当前用户”按钮，可以快速将当前 Windows 用户添加到 SQL Server 的 Windows 身份认证用户中。若要添加其他用户，则使用“添加”按钮。“数据目录”选项卡中可以设置数据库文件保存的默认目录。

说明：FILESTREAM 中的设置保持默认值即可。在本书第 19 章数据存储与索引中将专门对该功能进行详细讲解。

（12）单击“下一步”按钮，进入分析服务的配置界面。使用同样的方法为该服务配置用户和数据目录。

（13）单击“下一步”按钮，进入报表访问的配置界面。该界面提供了 3 个单选框用于用户选择。如果需要集成 SharePoint 的报表服务，则选择“安装 SharePoint 集成模式默认配置”选项。否则使用默认值选项即可。

（14）单击“下一步”按钮，进入 SQL Server 2012 新增的分布式重播控制器的配置界面，如图 1.13 所示。单击“添加当前用户”按钮，可以快速将当前 Windows 用户添加到 SQL Server 的 Windows 身份认证用户中。若要添加其他用户，则使用“添加”按钮。

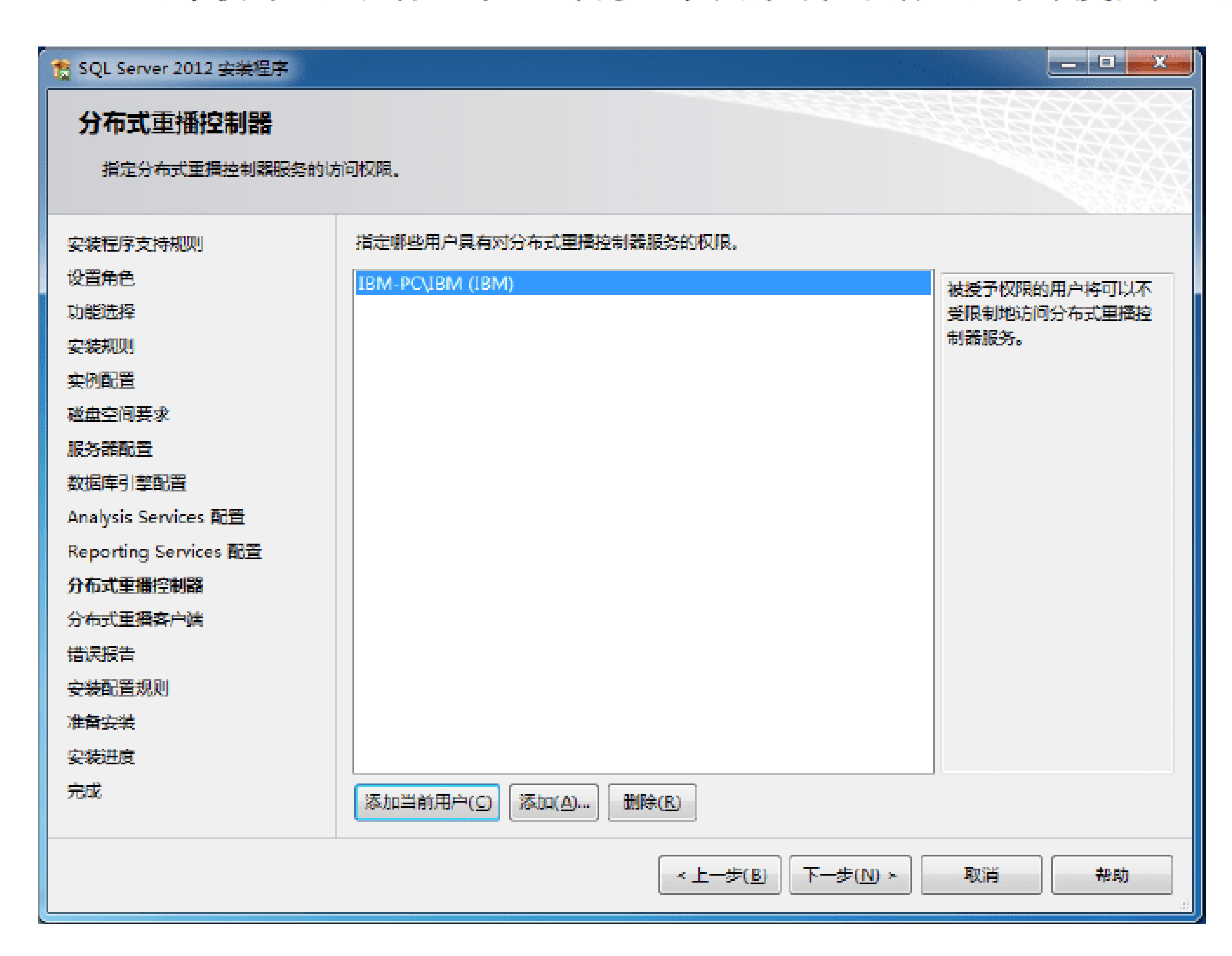


图 1.13　分布式重播控制器配置界面

（15）单击“下一步”按钮，进入分布式重播客户端的配置界面，如图 1.14 所示。在此界面中，填入控制器的名称，这里添加的是 kzq。

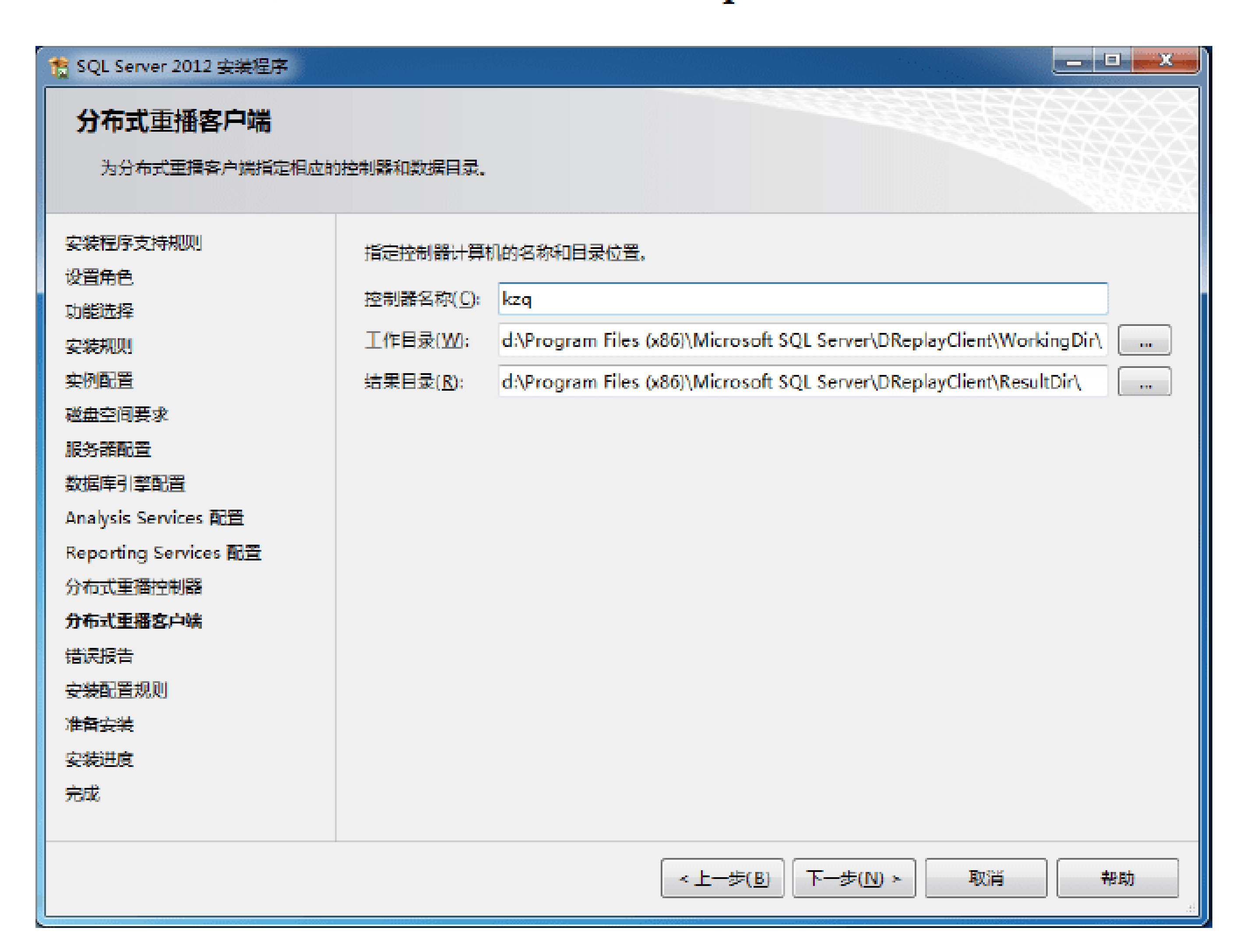


图 1.14　分布式重播客户端配置界面

（16）单击“下一步”按钮，系统将检查前面的配置是否满足 SQL Server 的安装规则。如果规则没有全部通过，则根据提示修改数据库或服务器中的对应配置，直到全部通过。

（17）继续单击“下一步”按钮直到“安装”按钮出现。然后单击“安装”按钮，SQL Server 2012 将按照向导中的配置将数据库安装到计算机中。在数据库安装完成后向导将显示成功安装的页面，至此 SQL Server 2012 顺利安装完成。

在 SQL Server 2012 安装完成后数据库服务将自动启动。打开 Windows 任务管理器，可以找到一个 sqlserver.exe 的进程。打开 Windows 的服务列表，可以找到服务 SQL Server（MSSQLSERVER）。其状态为“已启动”，启动类型为“自动”，如图 1.15 所示。通过这两种方式都可以看到数据库服务已经成功安装运行。

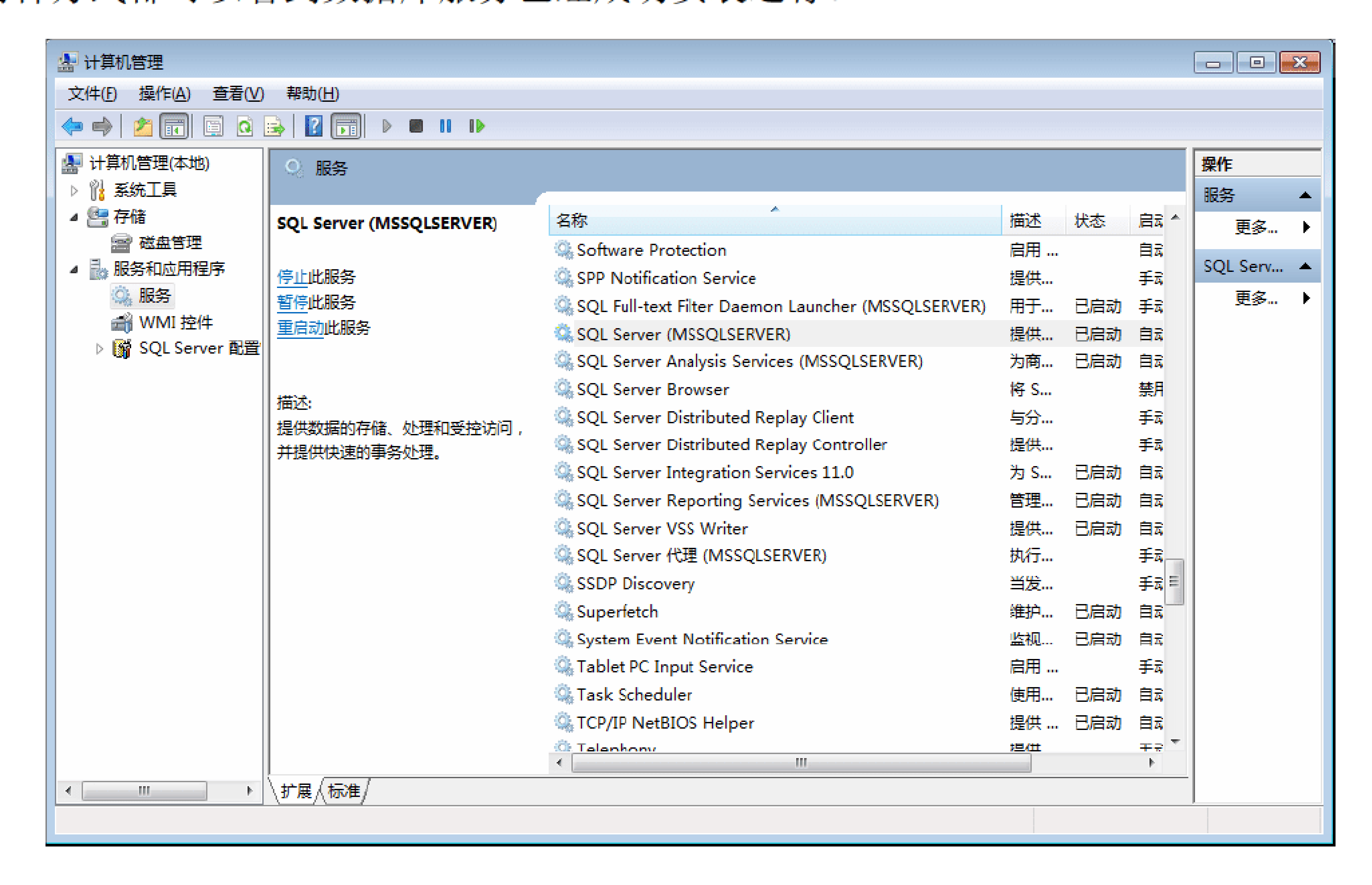


图 1.15　SQL Server 的服务

1.4　使用 SQL Server Management Studio

在 SQL Server 2000 中有企业管理器、查询分析器和 OLAP 分析管理器等管理工具用来对数据库进行管理。在使用中经常要在企业管理器和查询分析器中不断切换。在 SQL Server 2005、2008 版中将所有的操作集成到一个界面中，这就是 SQL Server Management Studio（简称 SSMS）。而 SQL Server 2012 继承了 SQL Server 2008 的操作风格，同样是使用 SSMS 来操作和管理数据库。

1.4.1　SQL Server Management Studio 简介

在正确安装 SQL Server 2012 后，Windows“开始”菜单下的程序列表中就会出现 Microsoft SQL Server 2012 的快捷方式，选择 SQL Server Management Studio 命令便可启动

SSMS。SSMS 启动后将弹出登录窗口如图 1.16 所示。

在此需要连接的服务器类型是数据库引擎，而服务器的名称就是安装运行了数据库服务的计算机的机器名或 IP 地址，该名由 SSMS 自动查找带出，如果在安装数据库时使用的不是默认实例，而是使用了实例名，那么服务器名称中还要包括实例名。比如服务器名称“127.0.0.1\SQLEXPRESS”就是连接本机的 SQLEXPRESS 实例。身份验证使用 Windows 身份验证，如果在安装数据库时配置了 sa 的登录密码，那么可以选择 SQL Server 身份认证，在用户名中输入 sa，然后输入配置的密码，单击“连接”按钮后，SSMS 将连接到指定的服务器。

连接到服务器后 SSMS 的总体界面如图 1.17 所示。SSMS 采用微软统一的界面风格。窗口最上面两排是菜单栏和工具栏，左侧是对象资源管理器窗口。所有已经连接的数据库服务器及其对象将以树状结构显示在该窗口中。中间区域是 SSMS 的主区域，SQL 语句的编写、表的创建、数据表的展示和报表展示等都是在该区域完成。主区域采用选项卡的方式在同一区域实现这些功能。右侧是属性区域，主要用于查看和修改某对象的属性作用。在图 1.17 中，属性区域自动隐藏到窗口最右侧，用鼠标移动到属性选项卡上则会自动显示出来。



图 1.16　SSMS 的登录窗口

图 1.17　SSMS 界面

注意：SSMS 中各窗口和工具栏的位置并不是固定的。用户可以根据自己的喜好将窗口拖动到主窗体的任何位置，甚至悬浮脱离主窗体。

1.4.2　使用 SSMS 打开表

在对象资源管理器中展开数据库，若按照 1.3.3 节中的步骤安装数据库，一般情况下可以看到 4 个系统数据库和两个用户数据库，如图 1.18 所示。

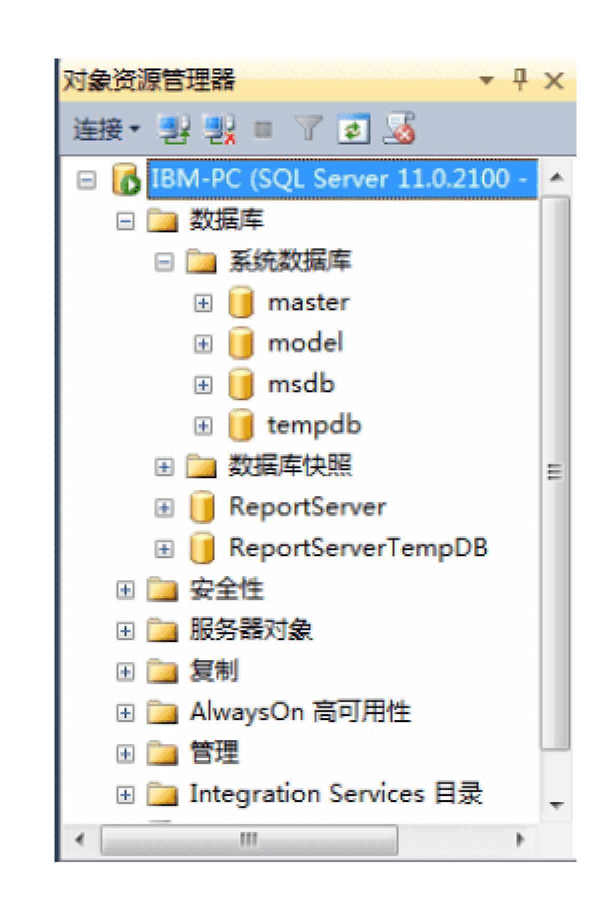


图 1.18　展开数据库

其中“系统数据库”节点下有 4 个数据库，在接下来的章节中会详细介绍这 4 个数据库。除了系统数据库外，还有两个数据

库，别是 ReportServer 和 ReportServerTempDB。ReportServer 和 ReportServerTempDB 是报表服务中使用的数据库。在 SQL Server 2005 中有 AdventureWorks 和 AdventureWorksDW 这两个 SQL Server 中自带的示例数据库。但是 SQL Server 2012 中并没有，必须从网站 http://msftdbprodsamples.codeplex.com/releases/view/55330 下载安装。在本书的大部分示例中都在 AdventureWorks2012 中操作，读者可以参考 1.7 节的内容先安装示例数据库。

继续展开 AdventureWorks2012 数据库下的表，可以看到该数据库下的所有表。现在需要查看某个表中的数据，比如查看 Person.AddressType 表中的数据时，可以在该表上右击，选择“打开表”选项，SSMS 将在主区域新建一个新的选项卡，并将该表的所有数据显示在该选项卡中，如图 1.19 所示。

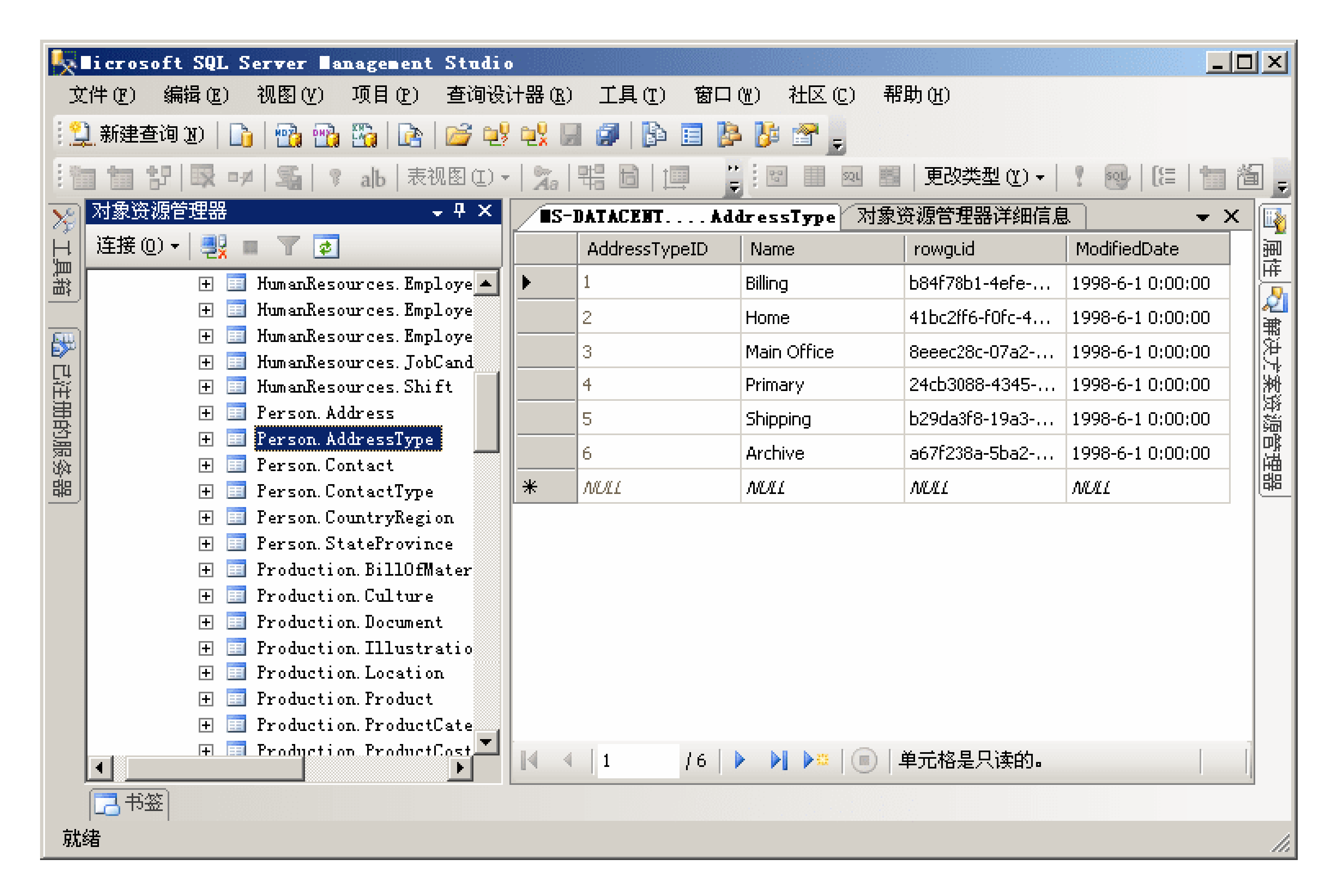


图 1.19　在 SSMS 中打开一个表

数据表下面的一行数据显示了该表有 6 行数据，而 AddressTypeID 列是灰色的，则表明该列是只读的。若在该选项卡中对该表添加或修改数据，SSMS 将会把更改提交到数据库系统中。

注意： 当表中有只读列时，若曾经将光标定位到只读列上，再打算修改该表中的数据时将无法使用中文输入法输入中文。这是 SSMS 的一个 Bug，到目前为止尚未得到解决。只有通过复制粘贴来输入中文，或者取消表中只读列的只读属性中文输入法，才可以正常输入中文。

在打开表的情况下单击工具栏中的显示 SQL 窗格按钮，SSMS 将在表结果的上面显示打开当前表结果所使用的 T-SQL 命令。当然，用户也可以修改其中的 T-SQL 语句得到需要的查询结果。在 SQL 窗格中修改 T-SQL 语句后需要单击工具栏中的执行 SQL 按钮，表中的内容将显示新的查询结果。

1.4.3　在 SSMS 中使用 T-SQL

SSMS 的主区域除了用来显示表数据和修改表数据外，还有一个十分重要且常用的功能，那就是编写 T-SQL 脚本。

SQL（Structured Query Language，结构化查询语言），是对关系数据库操作的公共语言。而 T-SQL 是 Transact-SQL 的简写，在此指的是 SQL Server 使用的 SQL 语言。若无特别说明，本书中的 SQL 语句指的都是 T-SQL 语句。在此主要介绍在 SSMS 中使用 T-SQL。关于 T-SQL 的详细内容，将在后面的章节中进行详细的介绍。

SSMS 支持对大多数数据库对象如表、视图、同义词、存储过程、函数和触发器等生成操作 SQL 语句，该功能减少了开发人员反复编写 SQL 语句的工作，大大提高了开发人员的工作效率。比如需要生成查询表 Person.AddressType 的 SQL 语句，只需要在该表上右击，选择“编写表脚本为”|“SELECT 到”|“新查询编辑器窗口”命令，如图 1.20 所示。SSMS 会在新选项卡中生成 Person.AddressType 表的 SQL 查询语句，如代码 1.1 所示。

代码 1.1　生成的 SELECT 脚本

```
USE [AdventureWorks2012]
GO

SELECT [AddressTypeID]
      ,[Name]
      ,[rowguid]
      ,[ModifiedDate]
  FROM [Person].[AddressType]
GO
```

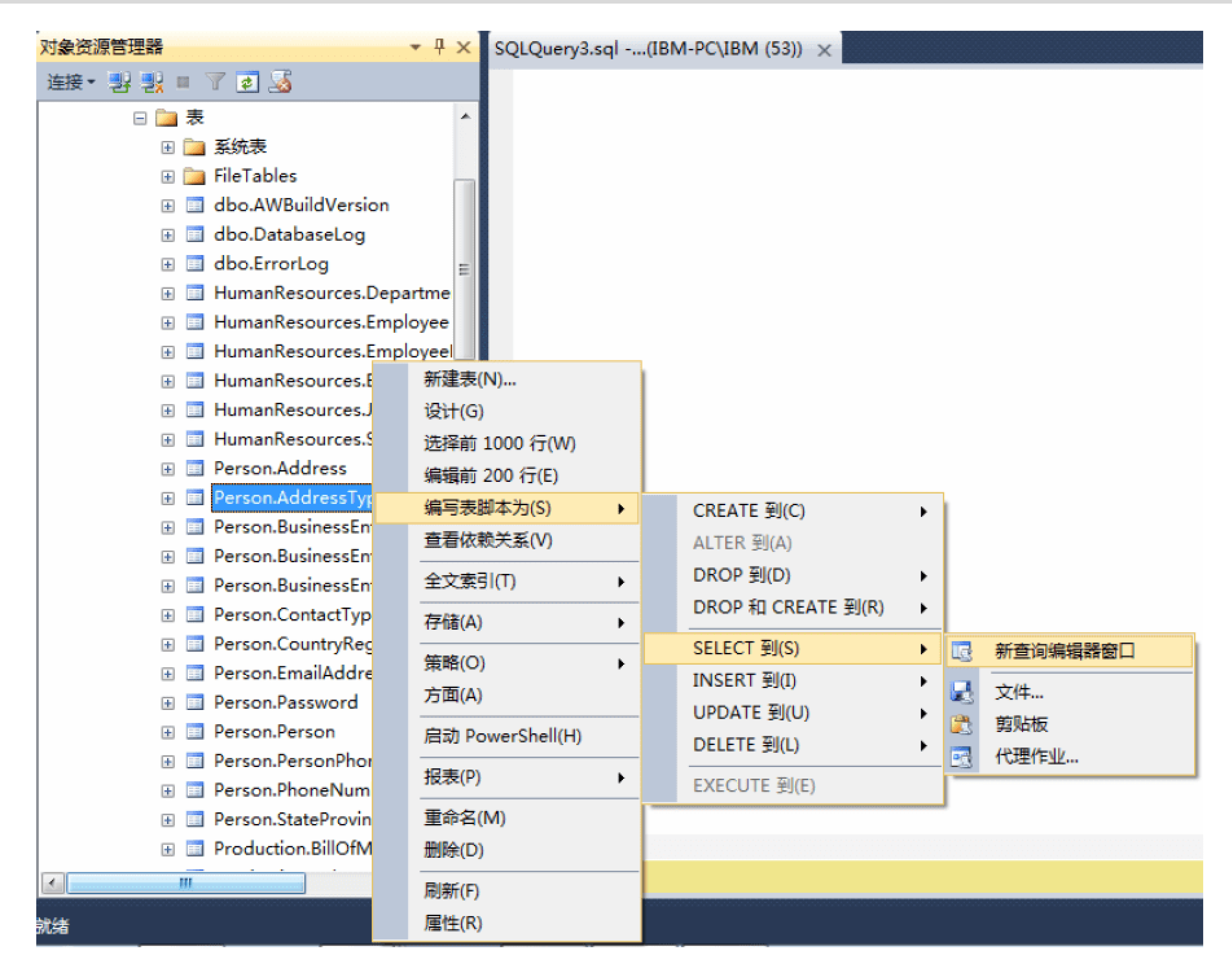


图 1.20　为表生产查询 SQL 语句

此时可以单击工具栏的“执行”按钮或直接使用快捷键 F5 键运行该 SQL 语句。运行后结果将在主区域中 SQL 语句下以表格的形式显示出来，如图 1.21 所示。表格结果下的状态栏还显示了一些和当前执行命令的相关信息。从左到右依次是数据库的版本、执行该命令的用户、执行命令的数据库、执行该命令所使用的时间和返回结果的行数。

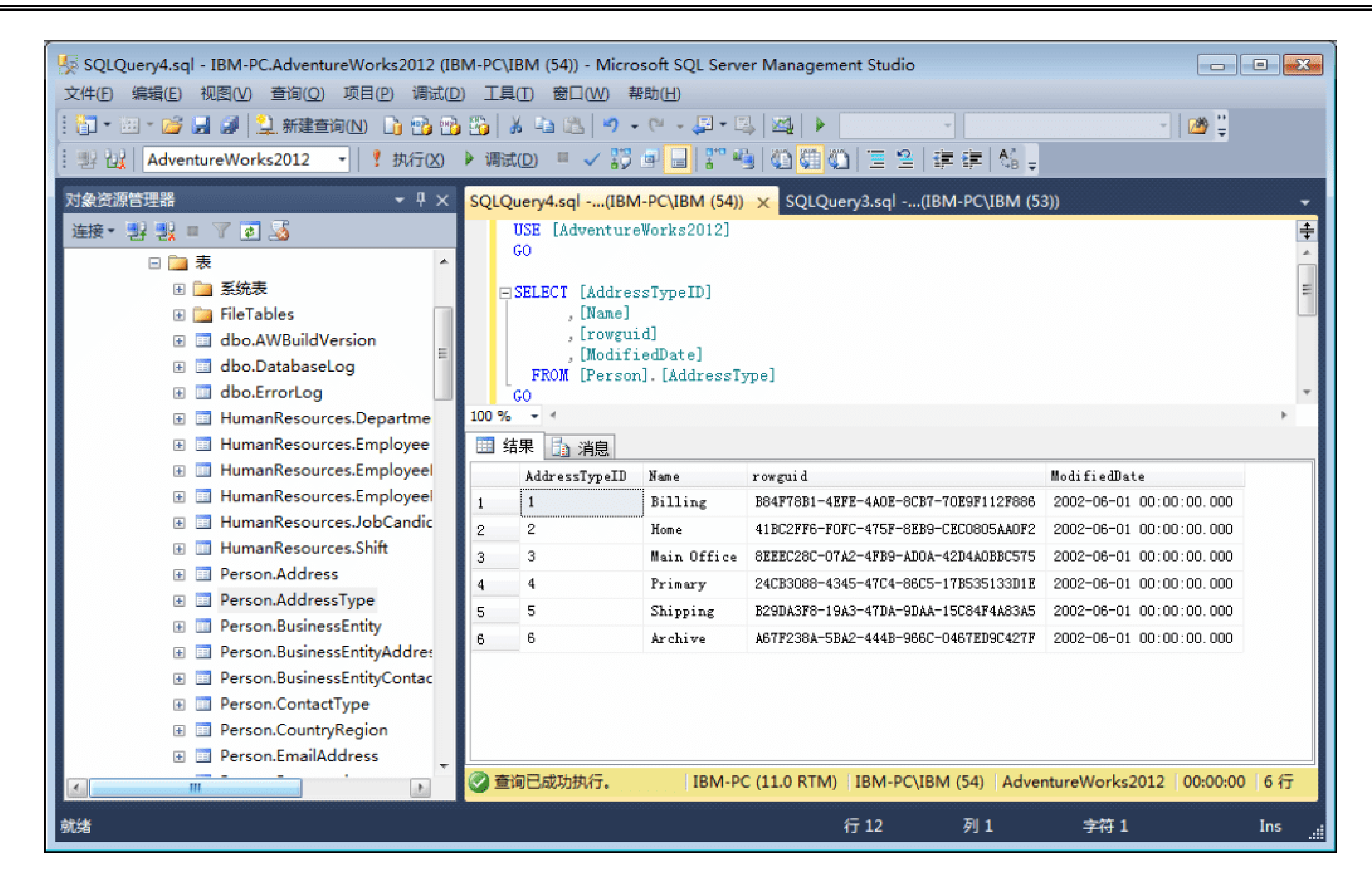


图 1.21　运行 SQL 语句

说明： 若用户在编辑器窗口中选中了部分脚本，SSMS 将只运行选中的 SQL 脚本；若编辑器窗口中用户没有选择任何脚本，SSMS 将运行该窗口中的所有 SQL 脚本。

正如图 1.18 所示，SSMS 除了提供生成查询语句外，还可以生成表的创建、删除、插入、更改和删除的 SQL 语句。若读者想自己编写 SQL 语句来运行，则可以先在对象资源管理器中选中要运行 SQL 语句的数据库或数据库下的对象，然后单击“新建查询”按钮或者使用快捷键 Alt+N，SSMS 将在主区域中新建一个空白编辑器窗口。读者可在此编写 SQL 语句，而工具栏的数据库下拉列表用于选择当前 SQL 语句所运行的数据库。

技巧： 数据库中的对象名并不需要通过键盘输入，用户可以将需要的对象名从左侧的对象资源管理器中用鼠标拖动到编辑器窗口中。这样既减少了用户的输入，也避免了输入拼写错误的情况发生。

1.4.4　使用 SSMS 管理服务器和脚本

当所操作和管理的数据库服务器较多时，对每个数据库服务器的操作都需要通过在登录窗口中输入数据库地址、用户名和密码来登录，这将是非常费时且非人性化的操作。为了管理多台数据库服务器，SSMS 允许将数据库服务器保存到“已注册的服务器”列表中。以后要对哪台服务器进行操作，只需在列表中双击该服务器便可连接登录到该服务器上，无需再输入用户名密码。

在“视图”菜单中选择“已注册的服务器”选项，SSMS 将会弹出“已注册的服务器”列表的窗口，如图 1.22 左上角所示。用户可以将所有使用的数据库服务器都添加到该列表中。另外，用户还可以对这些服务器进行分组以方便管理。对于已注册的服务器，用户只

需要通过双击便可连接到该服务器上。

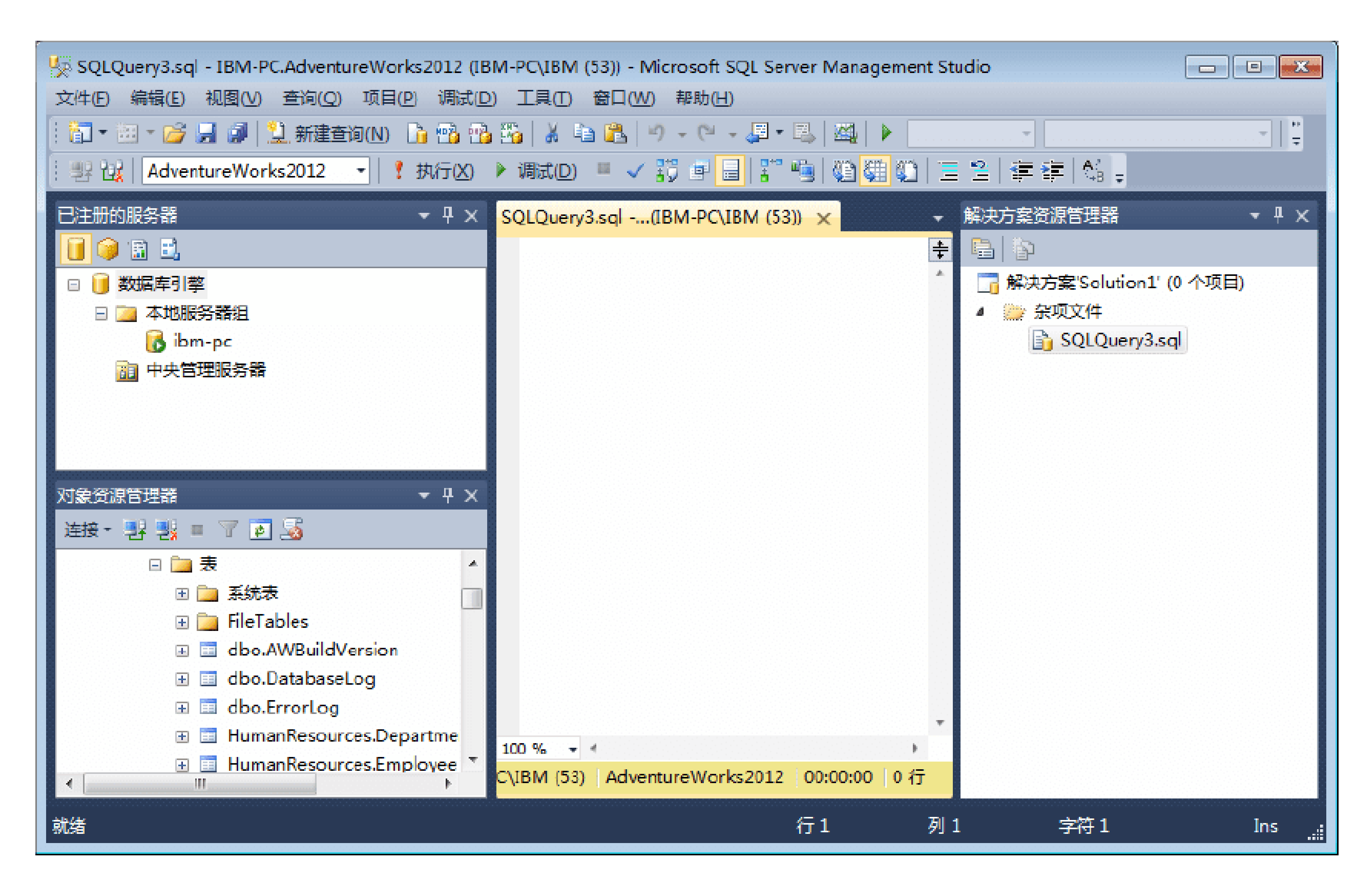


图 1.22　已注册服务器和解决方案资源管理器

在长期的数据库操作和维护过程中，对于一些常用的数据库操作脚本用户需要将其保存到硬盘上以便下次执行相同的数据库操作时再次使用。但是把脚本零散地保存到硬盘上不仅查找不方便而且也不便于以后的管理，为此 SSMS 提供了解决方案资源管理器来对脚本进行统一的管理。

在“视图”菜单中选择“解决方案资源管理器”选项，SSMS 将打开解决方案资源管理器。一个解决方案是由多个 SQL Server 脚本项目组成，而每个脚本项目下包含了 SQL 脚本和脚本执行所使用的连接，如图 1.22 所示。

用户可以将常用的脚本添加到解决方案中。在解决方案的脚本执行后，SSMS 会自动将该脚本执行的连接添加到解决方案中。使用解决方案资源管理器后用户不再需要到文件夹中一个一个地查找打开需要的脚本，只需要打开该解决方案并双击需要打开的脚本文件即可。同时用户也不用再去选择该脚本是在哪个数据库中执行，解决方案会自动将执行的数据库修改为上次执行该脚本的数据库。

1.5　SQL Server 2012 的其他工具

SSMS 的功能特别强大，在此不一一介绍。在后面的章节中将会继续介绍其他的功能。本节主要讲解除 SSMS 外 SQL Server 2012 自带的其他常用的工具。

1.5.1　使用配置管理器配置数据库

SQL Server 配置管理器主要用于管理 SQL Server 的服务、网络配置和客户端配置。

选择“开始” | Microsoft SQL Server 2012 | “配置工具” | “SQL Server 配置管理器”命令，系统将启动 SQL Server 配置管理器（SQL Server Configuration Manager），其界面如

图 1.23 所示。

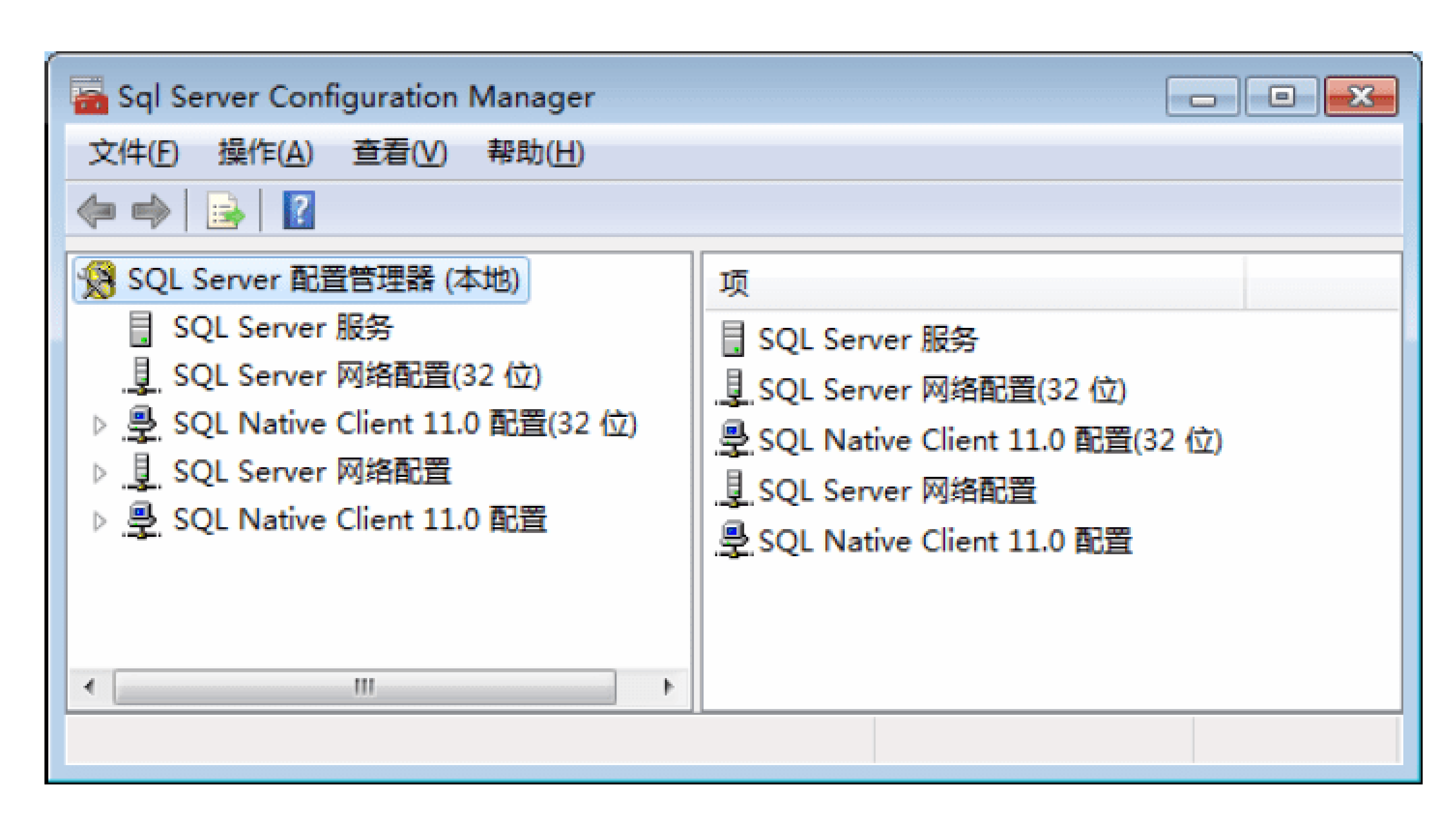


图 1.23　SQL Server 配置管理器的主界面

1. 管理SQL Server 2012服务

在配置管理器左窗口中单击“SQL Server 服务”链接，配置管理器将在右窗口以列表的形式展示当前计算机中所有安装的 SQL Server 2012 服务，以及服务的状态、启动模式、登录身份、进程 ID 和服务类型，其界面如图 1.24 所示。

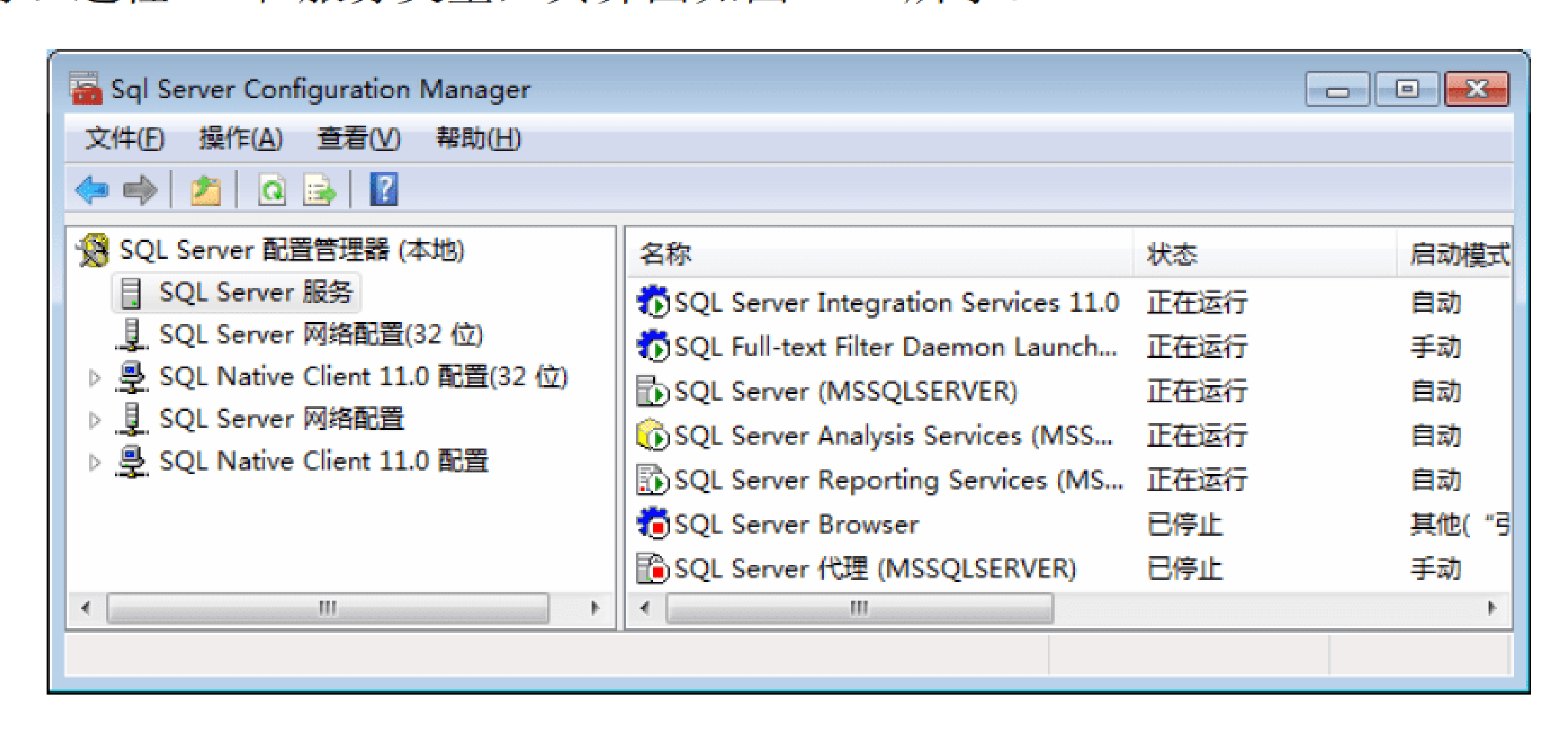


图 1.24　配置管理器中的 SQL Server 2012 服务

在完成安装 SQL Server 2012 企业版的情况下，SQL Server 提供以下服务。

- SQL Server Integration Services 集成服务是商务智能中的一部分。主要用于数据收集转换和数据仓库的建立。
- SQL Server FullText Search 全文检索服务，主要用于大量文本的检索。
- SQL Server 数据库服务提供基本的数据库运行支持。
- SQL Server Analysis Services 分析服务是商务智能的一部分，主要用于数据挖掘和 OLAP 分析等。
- SQL Server Reporting Services 报表服务用于报表的实现。
- SQL Server Browser SQL 浏览器主要用于多实例的网络支持。
- SQL Server Agent SQL 代理主要用于定时运行数据库作业。

服务名称后面的括号内容是该服务对应的 SQL Server 实例。如图 1.24 中的

MSSQLSERVER 就是默认实例名，而 SQL Server Integration Services 和 SQL Server Browser 后没有跟实例名是由于这两个服务是与实例无关的。也就是说，无论在一台计算机中安装了多少个 SQL Server 实例，这两个服务都各只有一个。

在列表中选中某服务后右击，可以将该服务启动、停止或重新启动。右击后选择“属性”选项或者双击某服务，系统将弹出该服务的属性对话框，如图 1.25 是 SQL Server（MSSQLSERVER）服务的属性对话框。

在该属性对话框中可以进一步修改服务的登录身份、启动模式和其他高级选项。

2. SQL Server 2012网络配置

在配置管理器的左窗口展开“SQL Server 2014 网络配置”节点，配置管理器会列出当前计算机的所有 SQL Server 实例。在图 1.24 中可以看出，该节点有一个链接“MSSQLSERVER 的协议”，这说明当前计算机中只有一个默认命名实例。

单击“MSSQLSERVER 的协议”链接，配置管理器的右窗口将列出该实例下的所有协议和协议的状态，如图 1.26 所示。

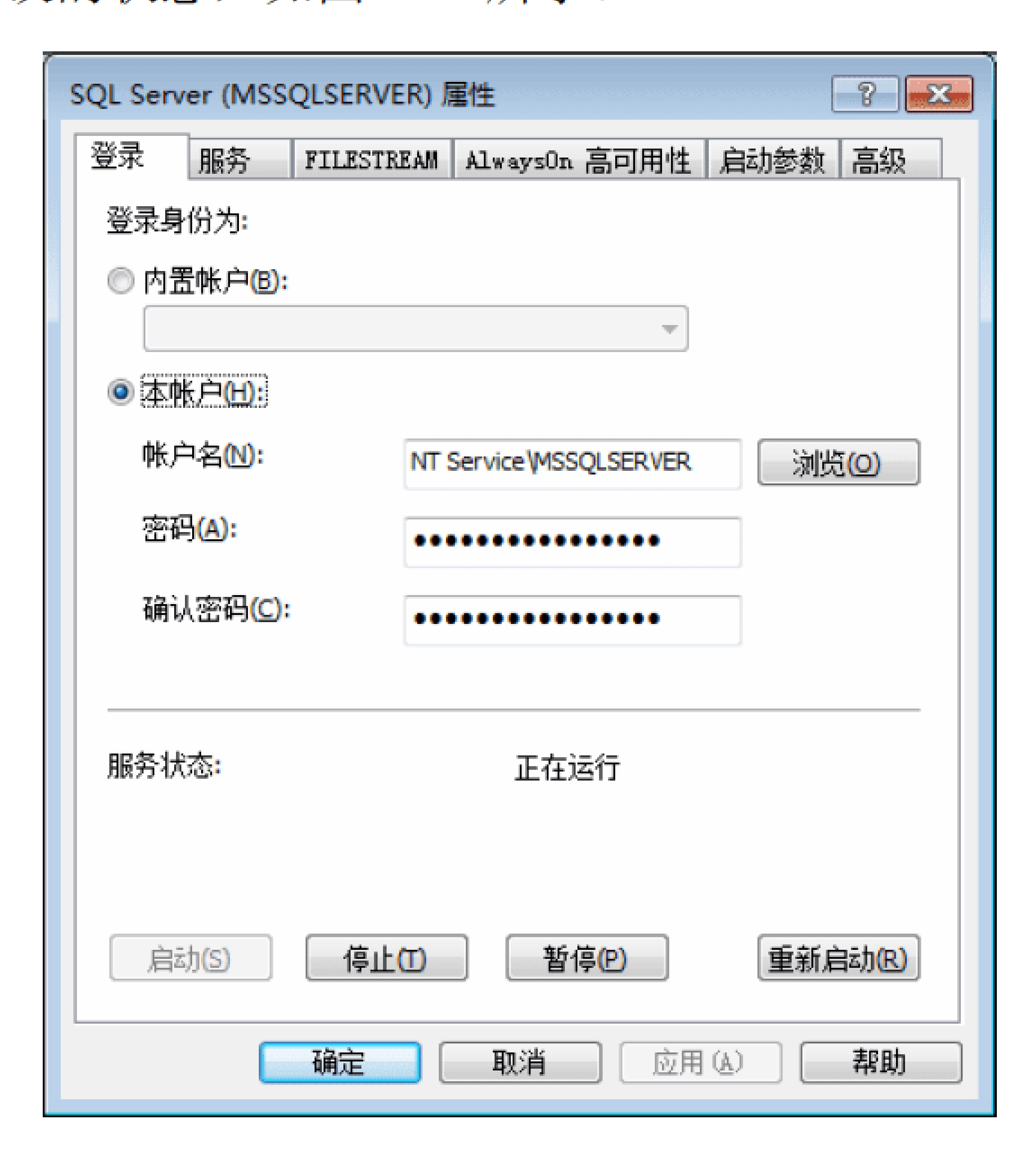


图 1.25　“SQL Server（MSSQLSERVER）属性”对话框

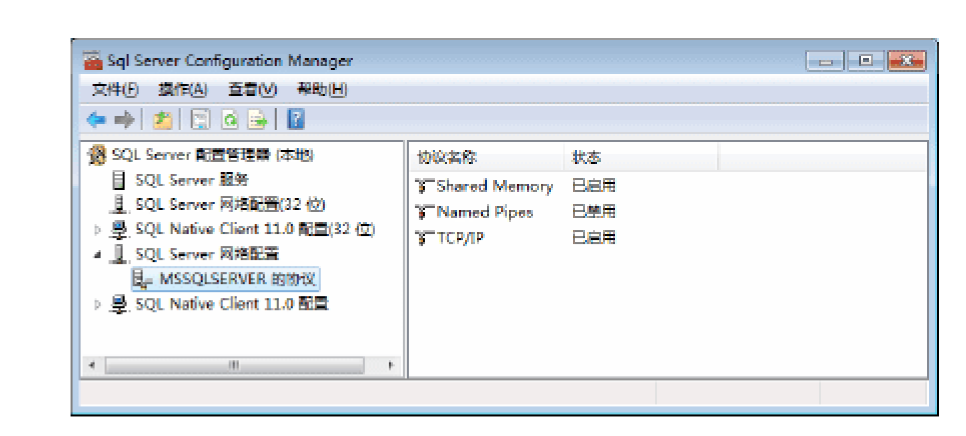


图 1.26　SQL Server 的协议

在 1.2.2 节中，已经介绍了 SQL Server 2012 支持的 3 种标准协议。配置管理器便可以实现对这 3 种协议的配置。用户可以通过右击某协议来选择打开或关闭该协议。一般情况下，只需要打开共享内存协议和 TCP/IP 协议即可。

由于 TCP/IP 协议使用最为广泛而其他协议比较简单或很少使用，所以在此只介绍 TCP/IP 协议的配置。为了防止 SSMS 使用其他协议连接数据库，所以先在配置管理器中把其他协议关闭，只打开 TCP/IP 协议。

右击 TCP/IP，在弹出的快捷菜单中选择“属性”选项或直接双击 TCP/IP，系统将打开 TCP/IP 协议的属性对话框，如图 1.27 所示。

“协议”选项卡主要是对活动状态 IP 侦听的配置。其中，“全部侦听”指定 SQL Server

是否侦听所有绑定到计算机网卡的 IP 地址。如果设置为“否”，则使用每个 IP 地址各自的属性对话框对各个 IP 地址进行配置。如果设置为“是”，则“IPALL”属性框的设置将应用于所有 IP 地址。默认值为“是”。

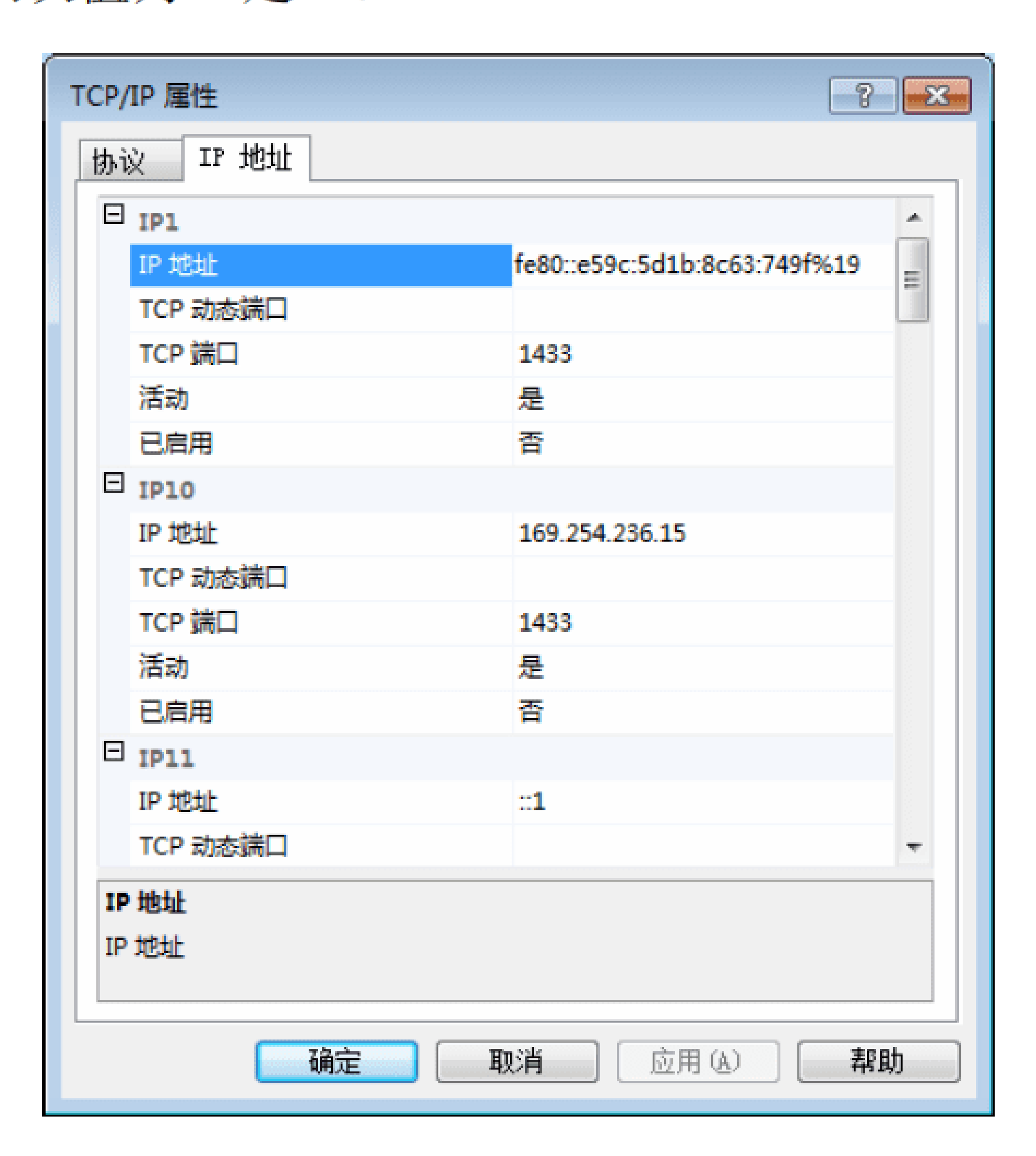


图 1.27　TCP/IP 属性配置

由于一台计算机可以使用多个 IP 地址，而在“IP 地址”选项卡中可以设置对于计算机上不同的 IP 地址使用不同的 TCP 端口。SQL Server 默认使用 TCP 端口 1433 进行通信，但是出于安全或其他因素的考虑，可以对通信端口进行修改。

比如，将 127.0.0.1 的 TCP 端口改为 1234 并将“已启用”选项设置为“是”，“全部侦听”选项设置为“否”。单击“确定”按钮并重启 SQL Server 服务。此时在 SSMS 下使用 127.0.0.1 作为数据库服务器地址将无法登录数据库。因为数据库的通信端口已经修改为 1234。修改端口后服务器地址的格式为“tcp:<server ip>,<port>”。在 SSMS 使用“tcp:127.0.0.1,1234”作为服务器地址才可以连接服务器。

以上只是对 127.0.0.1 这一个 IP 地址进行配置。若需要修改所有 IP 地址的通信端口，可以设置“全部侦听”为“是”，并修改 TCP/IP 属性中“IPALL”的 TCP 端口即可。

比如将 IPALL 的 TCP 端口修改为 12345，单击“确定”按钮并重启 SQL Server 服务。此时其他 IP 都必须使用 TCP 12345 端口才能通信。

3. SQL Native Client配置

SQL Native Client 配置主要包括客户端协议和别名。

客户端协议的配置界面与前面所讲的 SQL Server 2012 网络配置界面相同。不同之处在于此处是配置当前计算机中的客户端连接服务器时使用的协议。另外，还有一点不同之处就是客户端协议中可以配置协议的顺序。客户端将按照该顺序的协议去尝试连接服务器。若无法连接将使用下一个顺序的协议再次尝试，直到连接成功或所有协议都尝试完为止。

别名相当于 Oracle 中的 TNS 名，在配置管理器中新建的别名可以作为该计算机上客户端连接服务器时的服务器名称。在配置管理器的左窗口中单击“别名”链接，右窗口将

列出当前的所有别名。在右窗口中右击，选择“新建别名”命令将弹出新建别名的对话框，如图 1.28 所示。

在该窗口中输入别名的名称、端口号、服务器名或 IP 地址并选择协议后单击“确定”按钮，这样别名就建立好了。

再打开 SSMS，在登录窗口的“服务器名称”文本框中输入刚才新建的别名名称。单击“确定”按钮，SSMS 将使用别名中配置的服务器地址、端口和协议连接到对应的服务器。

图 1.28　新建别名

1.5.2　使用 SQL Server Profiler 跟踪数据库

除了对服务、协议和客户端的配置外，SQL Server 还提供了对数据库执行情况进行跟踪监视的工具 SQL Server Profiler。该工具是 SQL 跟踪的图形用户界面，用于监视 SQL Server Database Engine 或 SQL Server Analysis Services 的实例。用户可以捕获有关每个事件的数据，并将其保存到文件或表中供以后分析。

选择“开始”| Microsoft SQL Server 2012 |“性能工具”| SQL Server Profiler 命令，系统将启动 SQL Server Profiler。另外，在 SSMS 中选择“工具”| SQL Server Profiler 命令，也可以启动 SQL Server Profiler。

在 SQL Server Profiler 中选择“文件”|“新建跟踪”命令或者使用快捷键 Ctrl+N，系统将弹出登录窗口。该窗口与 SSMS 的登录窗口相似，输入需要跟踪的服务器名称、用户名和密码并单击“连接”按钮，Profiler 将连接到服务器并弹出跟踪属性窗口，如图 1.29 所示。

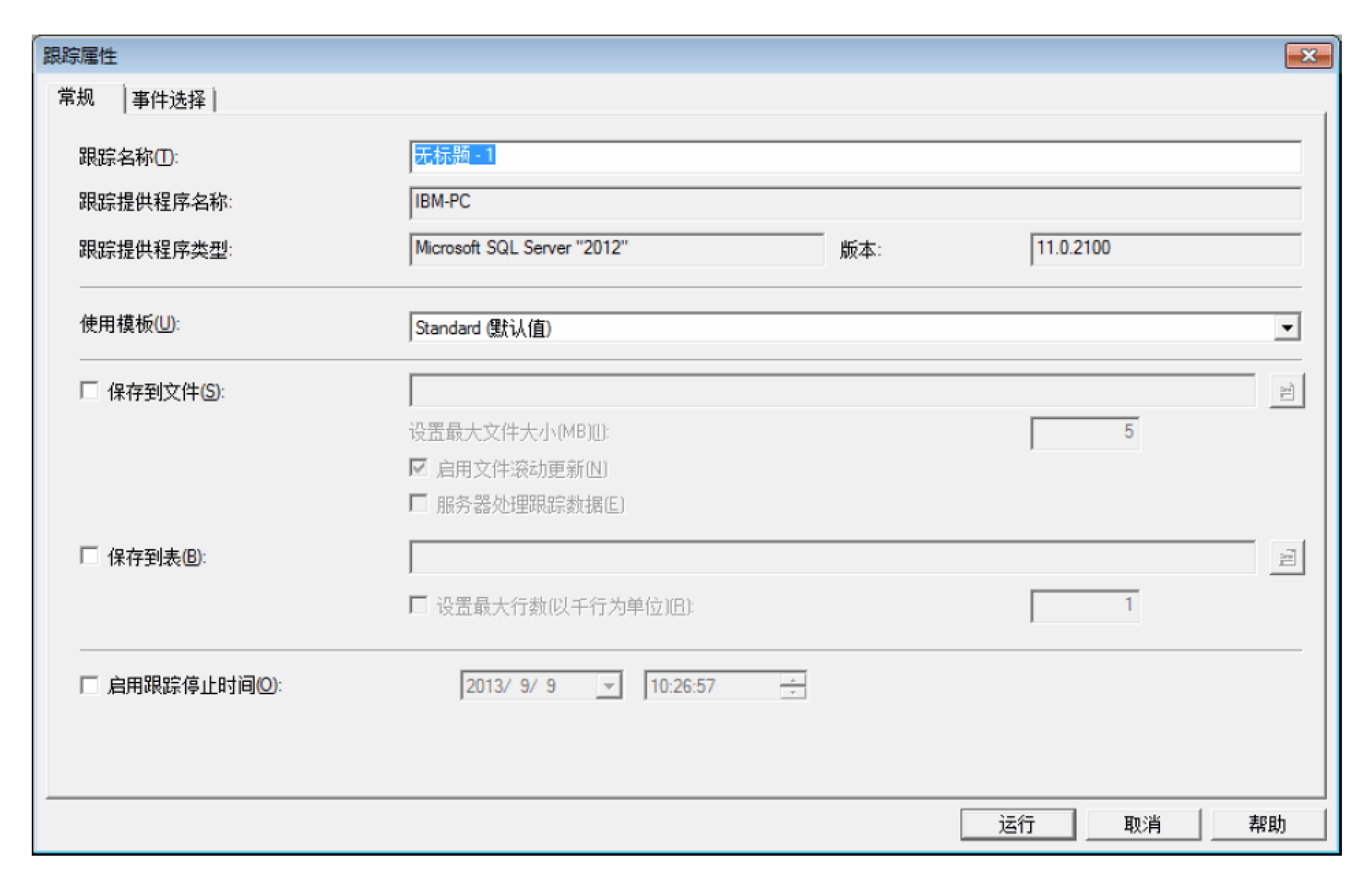


图 1.29　Profiler 跟踪属性

在常规选项卡中可以配置跟踪的名称、跟踪使用的模板、跟踪文件的保存方式和跟踪

的停止时间。一般为了对跟踪信息进行查询和统计，所以把跟踪信息都保存到另一个数据库系统中。

注意：有时出于安全、性能和其他因素的考虑会把跟踪信息保存到文件或者根本不保存，当停止跟踪以后可以再使用 Profiler 将跟踪信息导入到数据库表中进行查询和统计分析。

事件选择选项卡中列出了需要跟踪的事件以及事件的各个属性。用户可以选择本次跟踪所关心的事件，以及该事件的哪些属性。同时 Profiler 还提供了列筛选器，用户可以在其中定义筛选条件，Profiler 将只显示满足筛选条件的跟踪信息。关于各事件和事件属性笔者将在后面章节进行详细的讲解。此次跟踪属性可不做任何修改，然后单击“运行”按钮，Profiler 便开始对数据库进行监视。

在 Profiler 运行后使用 SSMS 打开被监视的数据并随便打开该数据库中的表。再切换到 Profiler 页面，便可以看到刚才 SSMS 执行数据库操作的所有 SQL 脚本，如图 1.30 所示。

在 Profiler 中单击工具栏的“停止所选跟踪”或单击“文件”菜单下的“停止跟踪”选项便可停止对服务器的跟踪。

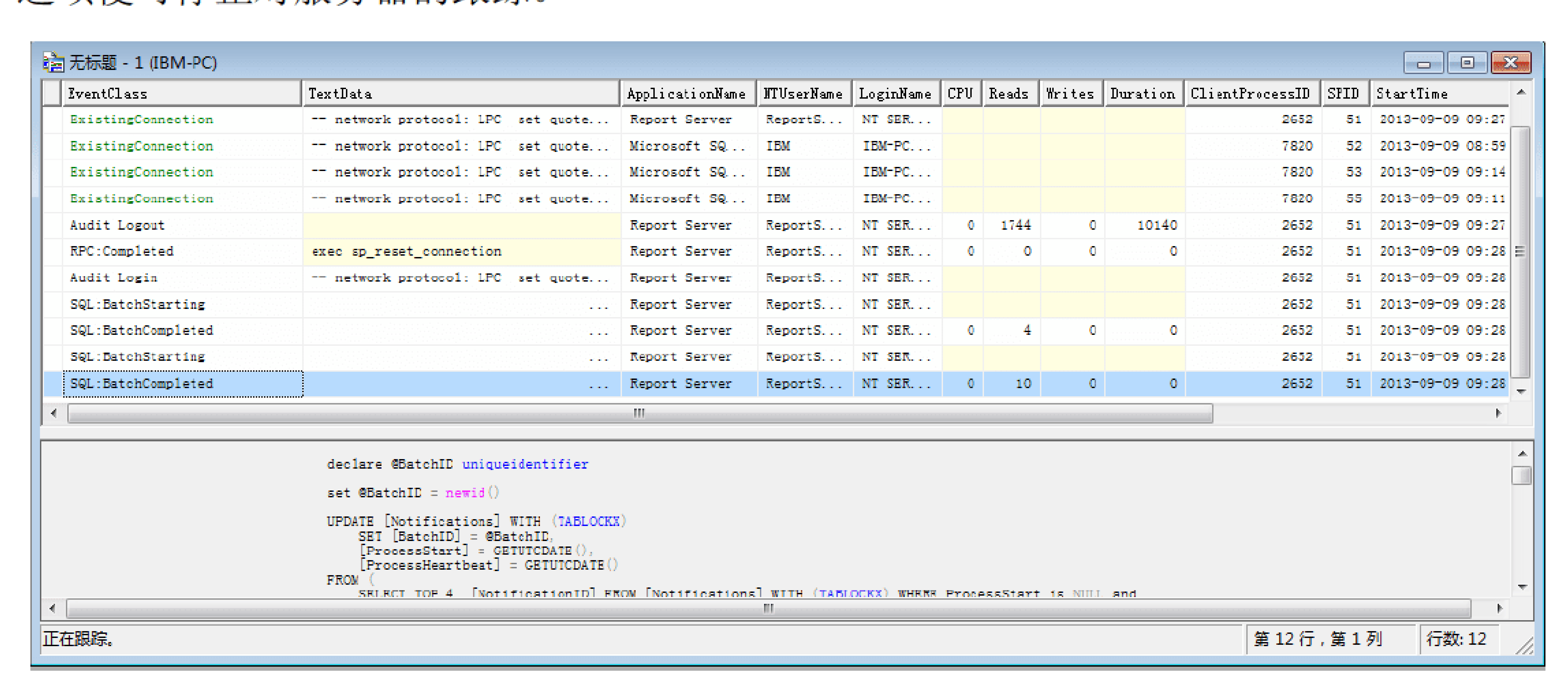


图 1.30　Profiler 跟踪的数据

1.5.3　使用 SQL Server 2012 联机丛书

对于 SQL Server 2012 这种功能强大而复杂的系统来说，帮助文档的地位无可替代。几乎没有人能够记住 SQL Server 中每个选项的含义，以及各种 T-SQL 语句。所以无论对于刚开始学习数据库的人还是资深的 DBA 来说，帮助文档都是必不可少的工具。

在 SSMS 中选择“帮助”菜单下的“如何实现”命令，或者直接使用 F1 键，系统将打开 SQL Server 2012 联机丛书。

说明：联机丛书是免费的，用户可以从微软的官方网站下载最新的联机丛书或直接查看联机丛书。

联机丛书的界面与 MSDN 帮助的界面相似，其界面如图 1.31 所示。由于在 SQL Server 2012 中取消了在直接安装帮助文档的介质，因此只能联机查看或是在网上浏览。



图 1.31　SQL Server 2012 联机丛书界面

1.6　SQL Server 2012 系统数据库简介

SQL Server 2012 默认自带了 4 个系统数据库。这 4 个系统数据库在整个数据库实例中担当着不同的角色。本节将主要介绍这 4 个系统数据库。

1.6.1　系统数据库 master——系统表的管理

master 数据库由一些系统表组成，这些系统表负责跟踪整个数据库系统安装和随后创建的其他数据库。master 数据库中记录了数据库的磁盘空间、文件分配和使用、系统层次的配置信息、端点和登录账号等信息。

如果 master 数据库不可用，则 SQL Server 无法启动。由于 master 数据库对系统来说至关重要，所以随时都应该保存一个其当前环境的备份。对数据库进行操作，比如创建修改或删除数据库、改变服务器配置值或者添加修改登录账号的操作之后，都应该备份一次 master 数据库。

由于 master 数据库的特殊性，系统对 master 做了一定的限制。用户不能在 master 数据库中执行下列操作。

- 添加文件或文件组。
- 更改排序规则。默认排序规则为服务器排序规则。
- 更改数据库所有者。master 归 dbo 所有。
- 创建全文目录或全文索引。
- 在数据库的系统表上创建触发器。
- 删除数据库。

- ❑ 从数据库中删除 guest 用户。
- ❑ 参与数据库镜像。
- ❑ 删除主文件组、主数据文件或日志文件。
- ❑ 重命名数据库或主文件组。
- ❑ 将数据库设置为 OFFLINE。
- ❑ 将数据库或主文件组设置为 READ_ONLY。

由于 master 数据库是系统数据库，而且在数据库变更时需要进行及时的备份，所以用户不应该把其他数据库对象放在该数据库中。

注意： master 数据库是数据库系统的默认数据库。用户使用 SSMS 登录数据库系统后新建的查询是针对 master 数据库的。用户可以在工具栏的下拉列表框中修改查询运行的数据库，或者在对象资源管理器中展开需要运行查询数据库再新建查询。

1.6.2　系统数据库 model——数据库的模板

model 数据库是一个模板数据库。当用户创建一个新的数据库时，系统将会复制 model 数据库作为新数据库的基础。如果希望每一个新的数据库在创建时还有某对象或权限，可以将这些对象或权限放在 model 数据库中，以后创建的数据库中将会包含这些对象或权限。

读者可以尝试在 SSMS 的对象资源管理器中右击“数据库”节点，选择“新建数据库”命令，系统将弹出“新建数据库”对话框。在数据库名称的文本框中输入 TestDB1，然后单击“确定”按钮，系统将新建一个数据库 TestDB1，如图 1.32 所示。

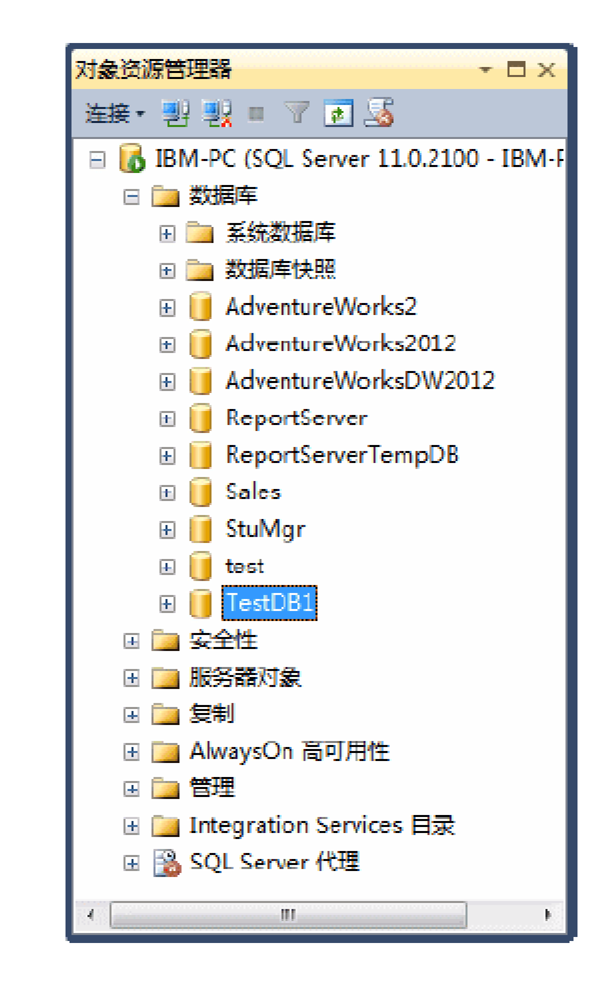


图 1.32　新建 TestDB1 的数据库

为了验证新建的数据库是以 model 数据库为模板建立的，将在 model 数据库中创建一个表 Table_1。

在对象资源管理器中依次展开“数据库”节点下的 model 数据库。右击“表”节点，选择“新建表”选项，SSMS 将在主区域打开一个新建表的选项卡。在列名中输入 abc 并单击工具栏的“保存”按钮，系统弹出“选择名称”对话框。默认是 Table_1，不做修改。单击“确定”按钮，此时 Table_1 表便在 model 数据库中建立好了。

在 Table_1 表建立好后，再使用创建 Test1 数据库的方式创建数据库 Test2。展开新建的 Test2 数据库表将可以看到其中包含表 Table_1。

model 数据库的更改只会影响在更改后新建的数据库。在该例子中，model 数据库新建的表 Table_1 只能影响之后创建的数据库 Test2，而之前创建的数据库 Test1 中则没有表 Table_1。

注意： 由于 model 数据库的更改会影响以后新建的数据库，所以读者不要在该数据库随便做更改。做完实验后，应该将 model 数据库恢复到以前状态。比如该例子中需要在实验结束后将 model 数据库中的 Table_1 表删除。

1.6.3　系统数据库 msdb——为 SQL Server 提供队列和可靠消息传递

在 1.5 节介绍配置管理器时，曾提到 SQL Server 服务中的最后一项 SQL Server 代理服务。代理服务主要用于数据库管理自动化，定时执行某些 SQL 脚本，定时进行数据库备份、复制任务，以及其他计划任务。SQL Server 代理服务将会使用到 msdb 数据库。在后面将讲到的 Service Broker 也会用到 msdb 数据库。msdb 为 SQL Server 提供队列和可靠消息传递。当不需要在数据库上执行备份和其他维护任务时，通常可以忽略 msdb 数据库。

在 SSMS 的对象资源管理器中可以访问 msdb 的所有信息，所以通常不需要直接访问该数据库的表。一般情况下，都不应该直接在 msdb 数据库表中添加删除数据，除非用户对自己的操作了解得十分透彻，或数据库专家允许用户这样做。

1.6.4　系统数据库 tempdb——临时工作区

tempdb 被用来作为一个工作区。tempdb 相对于其他 SQL Server 数据库一个很大的不同之处在于，每次 SQL Server 启动以后，系统将以 model 数据库为模板重新创建该数据库。tempdb 的这个特性使得用户不能将数据长期保存到该数据库中。在 SQL Server 再次启动时，tempdb 中的所有数据将不复存在。

当用户创建一个临时表时将使用到 tempdb 数据库。另外 tempdb 数据库还用于存储 SQL Server 在查询处理和排序时内部产生中间结果的工作表中，还可以用于维护在快照隔离级别和某些其他操作的行版本，填充静态游标和键集游标。在 SSMS 中新建查询输入语句：

```
SELECT 123 AS ID INTO #t
```

该语句的作用是创建了一个临时表#t。该表中只有一列 ID，并且只有一行数据 123。运行该语句后，再在对象资源管理器中展开 tempdb 数据库临时表节点，将可以看到创建的临时表#t，如图 1.33 所示。

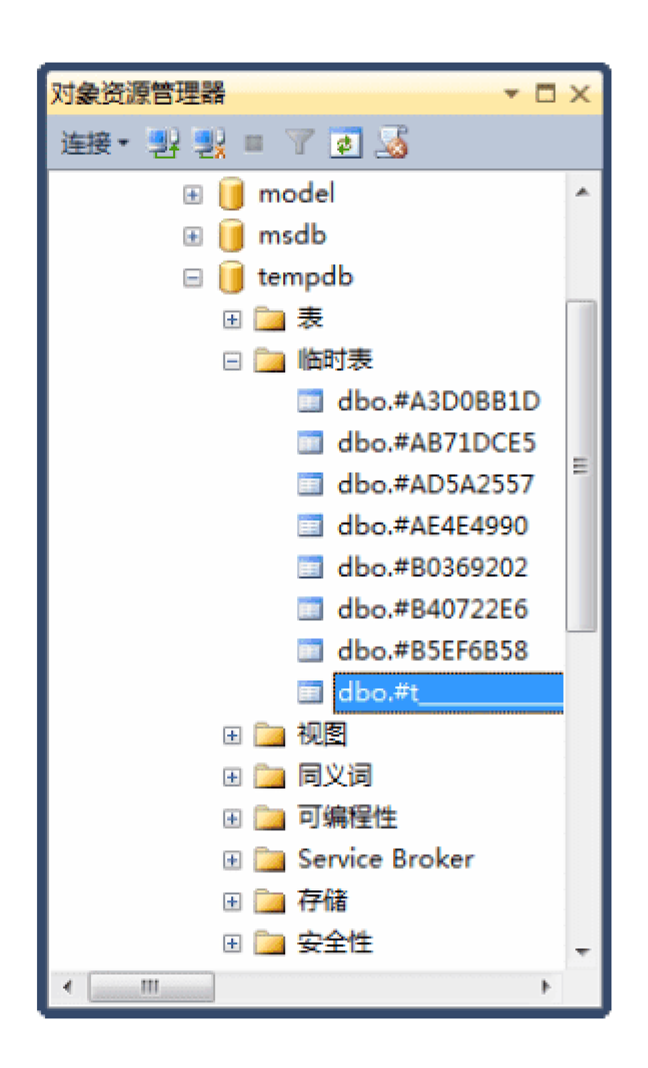


图 1.33　创建的临时表#t

由于 tempdb 的大小和配置对数据库性能优化至关重要，因此在本书的相关章节将对该数据库做更进一步的讲解。

1.7　示例数据库

微软为 SQL Server 提供了几个示例数据库，这些示例数据库中使用到了大量的 SQL Server 的特性。本书中的大部分示例都是基于示例数据库。本节将主要讲解示例数据库的安装并对各示例数据库进行简单的介绍。

1.7.1　安装示例数据库

SQL Server 2012 安装包中并没有提供示例数据库。由于本书中将会大量使用示例数据库进行举例，所以需要在安装完 SQL Server 后安装示例数据库。

SQL Server 2012 的示例数据库仍然是 AdventureWorks 的数据，它和 SQL Server 2008 大致相同，但添加了 SQL 2012 中的新特性。为了与 SQL Server 2008 区分，所以数据库名字叫做 AdventureWorks2012。具体安装及操作如下所述。

（1）登录网站：http://msftdbprodsamples.codeplex.com/releases/view/55330，这里提供了安装版的示例数据库和解压还原版的数据库。

（2）读者可根据自己的喜好选择下载安装版或者解压还原版，这里更倾向于使用完整的示例数据库实例附加。将以下文件下载。

- AdventureWorks2012 Data File 针对 SQL2012 新特性的 AdventureWorks 数据库的数据文件。
- AdventureWorksDW2012 Data File，商务智能时使用的数据库。

（3）附加数据库。将从网站上下载的数据库 AdventureWorks 2012.mdf 和 AdventureWorksDW2012.mdf 附加到当前的 SQL Server 2012 中。附加数据库的方法是在对象资源管理器中右击“数据库”节点，在弹出的右键菜单中选择“附加”选项，如图 1.34 所示。

在此界面中，找到相应的数据库的位置即可完成附加操作了。

1.7.2　示例数据库 AdventureWorks2012

AdventureWorks2012 是微软用户教育组虚拟了一个 Adventure Works Cycles 公司，并将该公司的数据以高度规范化的形式放在不同架构（Schema）中的示例数据库。

AdventureWorks2012 数据库与以往的 SQL Server 示例数据库的最大不同在于，该数据库中的表、视图和存储过程等数据库对象都是以不同架构的方式存放，用户拥有架构，而对象包含在架构中。表 1.3 列出了该数据库中的各架构和包含的对象，并列出每个架构中典型的表。

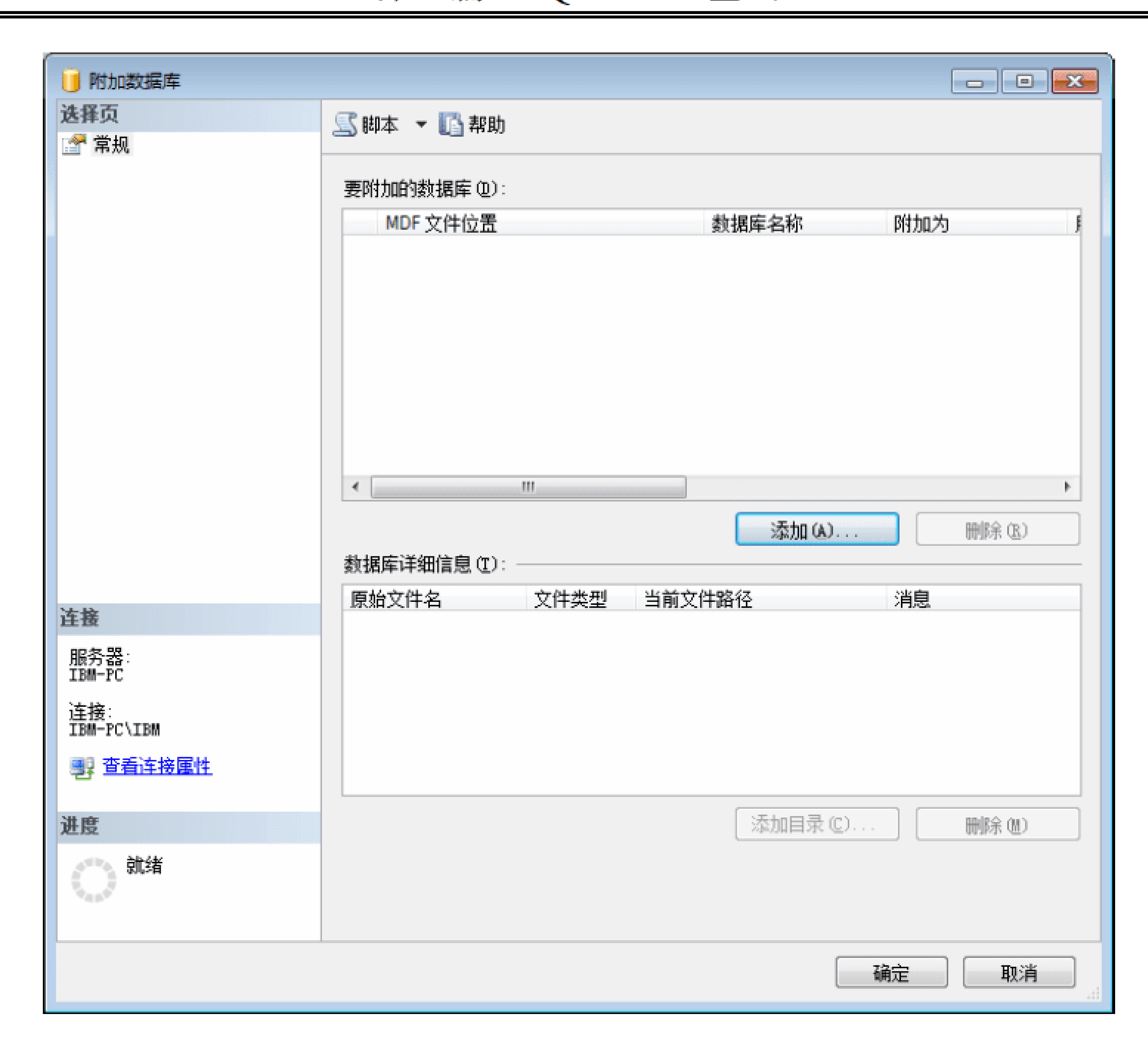


图 1.34　附加数据库界面

表 1.3　AdventureWorks2012 的架构说明

架　构	包含相关对象	示　例
HumanResources	Adventure Works Cycles 的员工	Employee 表 Department 表
Person	各个客户、供应商和雇员的名称和地址	Contact 表 Address 表 StateProvince 表
Production	由 Adventure Works Cycles 生产和销售的产品	BillOfMaterials 表 Product 表 WorkOrder 表
Purchasing	从该公司采购零件和产品的供应商	PurchaseOrderDetai 表 PurchaseOrderHeader 表 Vendor 表
Sales	与客户和销售相关的数据	Customer 表 SalesOrderDetail 表 SalesOrderHeader 表

除了架构外，AdventureWorks 还使用了 SQL Server 2012 的许多其他的特性。如 XML 数据类型、用户定义函数、数据库触发器和用户定义类型等。这些特性笔者在后面的章节中都会进行讲解。至于该数据库中每一个表的含义，以及表中每个字段的含义，读者可以翻阅 SQL Server 2012 自带的联机丛书。

1.7.3　示例数据库 AdventureWorksDW2012

AdventureWorksDW 包括了与数据仓库和 SQL Server 2012 数据仓库特性有关的特性。

该示例数据库包含两个主题区域：财务和销售。财务部分由财务和货币汇率组成；而销售部分由分销商销售、销售汇总、网上销售和销售配额组成。

本书在商务智能章节将主要以该数据库为基础进行讲解，此处不做详细介绍。

1.8　小　　结

本章从介绍 SQL Server 的发展历史开始，讲解了 SQL Server 2012 的特点、架构、安装，以及各种相关工具的使用。最后简单介绍了 SQL Server 2012 中的系统数据库和示例数据库。通过本章的学习，相信读者对 SQL Server 2012 的基本概念和操作已经有了一个初步的了解。

第 2 章　T-SQL 基础

T-SQL 是 Transact-SQL 的缩写，是使用 SQL Server 的核心。与 SQL Server 通信的所有应用程序都通过将 Transact-SQL 语句发送到服务器来实现相应的功能。本章将主要讲解 T-SQL 的一些基础知识。

2.1　T-SQL 简介

SQL（Structured Query Language，结构化查询语言）从发明到现在经过几十年的发展，已经成为关系数据库的标准语言。本节将介绍 SQL 的发展历史、分类和语法约定等知识。

2.1.1　SQL 背景

SQL 的最早版本是由 IBM 开发的，最初叫做 Sequel，是 20 世纪 70 年代早期作为 System R 项目的一部分实现的。1980 年它的名字变为 SQL。1979 年 Oracle 公司首先提供商用的 SQL。IBM 公司在 DB2 和 SQL/DS 数据库系统中也实现了 SQL。后来 SQL 在其他数据库产品中也得到支持。如今 SQL 已经确立了其作为标准关系数据库语言的地位。

1986 年，美国国家标准化组织（ANSI）和国际标准化组织（ISO）发布了 SQL 标准：SQL-86。1989 年，ANSI 发布了一个 SQL 的扩展标准：SQL-89。该标准的下一个版本是 SQL-92 标准，接着是 SQL:1999 标准。目前最新的标准是 SQL:2003。

各大数据库厂商为满足产品的某些特性以 SQL 标准为基础进行了延伸和扩展。Oracle 使用的 SQL 语言被称为 PL-SQL，而 SQL Server 则使用的是 T-SQL。T-SQL 基本上是根据 1992 年发表的 ISO 标准出台的，只是在 1999 年的标准上稍加修改。此外，微软还进行了各种私有的加强。

标准 SQL 和 T-SQL 之间有很多区别。如果在 SQL Server 上工作，那么使用这些私有的扩展是有好处的。由于许多 SQL Server 特性的本质问题，如果不使用非标准的命令，将会有很多强大的功能无法实现。如果用户想要看看自己的 SQL 是否符合标准，可以使用 SET FIPS_FLAGGER 命令。

2.1.2　SQL 语言分类

SQL 语言由于功能的不同被人为地划分为以下几部分组成。

- 数据定义语言（Data-Definition Language，DDL）：SQL DDL 提供定义关系模式、修改以及删除关系模式的命令，以 CREATE、ALTER、DROP 和 GRANT 等命令

为主。

- 数据操纵语言（Data-Manipulating Language，DML）：SQL DML 包括基于关系代数和元组关系演算的查询语言，还包括在数据库中插入、删除和修改元组的命令，以 INSERT、UPDATE、DELETE 和 TRUNCATE 语句为主。
- 数据查询语言（Data-Query Language，DQL）：SQL DQL 包括对数据的查询命令，即 SELECT 命令的语句。另外在很多情况下人们将 DQL 和 DML 混在一起，将 SELECT 命令划入 DML 语句。
- 数据控制语言（Data-Control Language，DCL）：SQL DCL 包括说明对关系和视图的访问权限的命令，以及对数据库操作事务的控制和对数据库实时监视等，以 COMMIT、ROLLBACK、GRANT 和 REVOKE 命令为主。

注意：GRANT 语句在 CREATE SCHEMA 语句中使用时为 DDL 语句，而在架构定义外给予用户额外的权限时为 DCL 语句。

2.1.3　语法约定

本书在描述 T-SQL 语法时使用的约定与 SQL Server 联机丛书中的语法关系图相同，其约定如表 2.1 所示。

表 2.1　语法约定

约　　定	用　　于
大写	Transact-SQL 关键字
斜体	用户提供的 T-SQL 语法的参数
粗体	数据库名、表名、列名等数据库对象名，以及必须按所显示的原样输入的文本
下划线	指示当语句中省略了包含带下划线的值的子句时应用的默认值
|（竖线）	分隔括号或大括号中的语法项。表示单选，只能使用其中一项
[]（方括号）	可选语法项。不要输入方括号
{}（大括号）	必选语法项。不要输入大括号
[,...n]	指示前面的项可以重复 n 次。各项之间以逗号分隔
[...n]	指示前面的项可以重复 n 次。每一项由空格分隔
[;]	可选的 T-SQL 语句终止符
<label>::=	语法块的名称。此约定用于对可在语句中的多个位置使用的过长语法段或语法单元进行分组和标记。可使用的语法块的每个位置由括在尖括号内的<label>标签指示

SQL Server 支持的对象引用全名是 server.database.schema.object。一般情况下，SSMS 中已经指定了 server 名和 database 名，所以一般会简写为 schema.object 的形式。如果该对象使用的是 dbo 架构，那么对对象的引用还可以进一步简写为 object 的形式。比如 AdventureWorks.dbo. DatabaseLog 对象就可以直接简写为 DatabaseLog 的形式。

2.2　基本的 SQL 语句

SQL 对数据库的基本操作主要是 CRUD 操作。CRUD 是 Create（创建）、Read（读取）、

Update（更新）和 Delete（删除）的缩写。本节将主要讲解如何使用 T-SQL 实现 CRUD 操作。

2.2.1　使用 SELECT 查询数据

最基本的数据查询语句是由 SELECT…FROM…组成。其中，SELECT 子句用于表示查询关注的结果，FROM 子句用于表示查询针对的表。用户可以在此把需要关注的属性列出来，若需要关注所有的属性可以使用“*”来表示，例如：

```
SELECT *
FROM loan
```

该语句用于查询 loan 表的所有列的数据。如果只需要查询 loan 表中的 loanID 和 loanMoney 两个字段，则每个字段之间使用“,”进行分隔。其查询的 SQL 脚本为：

```
SELECT loanID,loanMoney
FROM loan
```

除了直接返回查询结果外，SELECT 子句还可以对返回的结果进行数学运算和指定新属性名称。T-SQL 中使用 AS 命令来重命名表和字段。例如：

```
SELECT loanID,loanMoney+payMoney AS totalMoney,loanMoney*0.5 costMoney
FROM loan
```

注意：*在对表和字段等进行重命名时 AS 命令一般是可以省略的，loanMoney*0.5 costMoney 就相当于 loanMoney*0.5 AS costMoney。虽然 SQL Server 支持该简写，但是建议读者在写 SQL 时不要省略 AS，这样写出的代码更易读。*

在查询语句后跟 WHERE 子句用于设定查询结果的筛选条件。在 WHERE 子句中可以使用 AND、OR 和 NOT 运算符，另外还可以使用比较运算符“<”、“>”和“=”等。

为了简化 WHERE 子句，SQL 提供了 BETWEEN 比较运算符来说明一个值小于或等于某个值，同时大于或等于另一个值。若想要找出分数在 60～90 分的所有学生的名字，那么可以写成代码 2.1 的形式。

代码 2.1　WHERE 子句

```
SELECT StuName
FROM StudentScore
WHERE Score BETWEEN 60 AND 90 --WHERE 条件查询
```

在代码 2.2 中实现了对 Bothell 城市的所有用户所在国家名、州名，以及用户地址信息的查询。该查询中使用了 3 个表进行连接并对表进行了重命名。返回结果中也对返回列名进行了重命名以示区别不同的 Name。

代码 2.2　典型数据查询

```
SELECT cr.Name AS CountryRegion,sp.Name StateProvinceName,a.*
FROM Person.Address a
```

```
INNER JOIN Person.StateProvince sp --内联接
ON sp.StateProvinceID = a.StateProvinceID
INNER JOIN Person.CountryRegion cr
ON cr.CountryRegionCode = sp.CountryRegionCode
WHERE a.City='Bothell'
```

2.2.2 使用 INSERT 插入数据

INSERT 命令负责将数据插入到数据表中。INSERT 命令的基本语法格式为：

```
INSERT INTO <table name> [(<column name> [{,<column name>}])]
VALUES(<values>[,{<values>}])
```

其中表的列名是可选的。如果要为每一可写列指定值，则可以省略表的列名。在未指定列名的情况下，插入值的顺序必须与表中列的顺序对应。如果指定了列名则插入值的顺序要与指定列的顺序对应。

以 AdventureWorks2012 数据库中的 Person.ContactType 表为例，该表有 3 个字段，分别是 ContactTypeID、Name 和 ModifiedDate。其中，ContactTypeID 为主键并且为自增列，Name 列必须由用户输入，ModifiedDate 使用 GETDATE()函数作为默认值，记录了插入数据的时间。同时该表使用了触发器将每一行数据修改的时间记入 ModifiedDate。关于默认值和触发器将在第 3 章进行讲解，此处暂不讨论。若想要向 ContactType 表中插入一行数据可以参照代码 2.3 所示。

代码 2.3 使用 INSERT 插入数据

```
USE AdventureWorks2012
GO
INSERT INTO Person.ContactType(Name,ModifiedDate)
VALUES('Test1','2013-1-1')
GO
INSERT INTO Person.ContactType --不带列名的插入
VALUES('Test2','2013-1-1')
GO
INSERT INTO Person.ContactType(ModifiedDate,Name) --重新排列了列名的插入
VALUES('2013-1-1','Test3')
```

技巧：若要向表 Person.ContactType 中插入数据，可以在 SSMS 中右击该表，在弹出的快捷菜单中选择“编写脚本为”|“INSERT 到”|“新查询编辑器窗口”命令，系统将根据该表生成 INSERT 语句的模板，修改该模板即可。

INSERT 命令除了通过 VALUES 来指定要插入的数据外，还可以使用 SELECT 命令代替 VALUES 子句，将 SELECT 查询的结果集插入到表中。使用 SELECT 语句的关键和使用 VALUES 子句一样，那就是必须保证 SELECT 语句返回的结果集和所需的值一致，并且这些值要符合表的限制。

例如，现在要将 Person.AddressType 中的数据全部插入到 Person.ContactType 表中（这里只是为了举例，实际上这个操作也许不会发生），就可以使用 INSERT 和 SELECT 结合

进行操作。具体 SQL 脚本如代码 2.4 所示。

代码 2.4　INSERT 和 SELECT 结合

```
INSERT INTO Person.ContactType
(
    Name,
    ModifiedDate
)
SELECT at.Name,at.ModifiedDate
FROM Person.AddressType at
--将查询出的结果插入到表中
```

在使用 INSERT 和 SELECT 语句结合时，并不一定要将所有 SELECT 查询的结果插入到表中，可以使用 TOP 命令指定要插入的数据的数量。TOP 可以用于 SELECT 上指定查询的数量，也可以用于 INSERT 上指定插入的数量。例如，要将 Person.Address 中的前 3 行数据插入到 Person.ContactType 表中的 SQL 脚本，如代码 2.5 所示。

代码 2.5　带 TOP 的 INSERT 语句

```
INSERT TOP(3) INTO Person.ContactType --插入查询出的前 3 行结果
(
    Name,
    ModifiedDate
)
SELECT a.AddressLine1,a.ModifiedDate
FROM Person.Address a
--或者
INSERT INTO Person.ContactType
(
    Name,
    ModifiedDate
)
SELECT TOP 3 a.AddressLine1,a.ModifiedDate
FROM Person.Address a
```

2.2.3　使用 UPDATE 更新数据

UPDATE 命令用于更新数据库中的数据，利用 UPDATE 命令可以修改一列或多列中的一行或多行数据。UPDATE 命令的语句格式为：

```
UPDATE <table name>
SET <set clause expression> [{, <set clause expression>}...]
[WHERE <search condition>]
```

在 UPDATE 命令中 SET 子句是必需的，而 WHERE 子句是可选的。WHERE 子句的使用与 SELECT 命令相同。此处 WHERE 子句用于指定搜索条件，满足条件的数据才会被更新，未使用 WHERE 子句则更新所有记录。SET 子句中的<set clause expression>是一个占位符，使用<column name>=<value expression>形式指定一个或多个表达式。

例如要将前面添加到 ContactType 表中的数据进行更改，在原 Name 的前面添加“--”

即可。由于 ContactType 中原有 20 行数据，所以必须使用 WHERE 条件进行更新。具体 SQL 脚本如代码 2.6 所示。

代码 2.6　UPDATE 更新语句

```
UPDATE Person.ContactType
SET Name='--'+Name
WHERE ContactTypeID>20 --只更新前面插入的数据，原数据不变
```

除了使用 WHERE 条件来限定一个表的更新范围外，还可以使用包含 FROM 子句的 UPDATE 语句。例如，在 ContactType 中插入了 3 行 AddressType 的数据，现在需要将这 3 行数据进行更改，使 ContactType 中的 Name 保存的是 AddressType 的 rowguid 项。对于这种将一个表的数据更新到另外一个表的情况，就需要使用 FROM 子句。具体 SQL 脚本如代码 2.7 所示。

代码 2.7　使用包含 FROM 子句的 UPDATE 语句

```
INSERT TOP(3) INTO Person.ContactType  --插入 3 行数据
(
    Name,
    ModifiedDate
)
SELECT at.Name,at.ModifiedDate
FROM Person.AddressType at
GO
UPDATE Person.ContactType  --将 AddressType 的 rowguid 更新到 ContactType 的
                             Name 中
SET Name = at.rowguid
FROM Person.ContactType ct
INNER JOIN Person.AddressType at
ON ct.Name=at.Name
```

另外，UPDATE 语句与 INSERT 语句一样，也可以使用 TOP 子句来指定要更新的行数。例如，需要将 ContactType 中插入的第 2 行数据进行更改，使其 Name 为 ContactTypeID，修改脚本如代码 2.8 所示。

代码 2.8　使用带 TOP 的 UPDATE 语句

```
UPDATE TOP(2) Person.ContactType  --只修改第 2 行数据
SET
    Name = ContactTypeID
WHERE ContactTypeID>20
```

2.2.4　使用 DELETE 删除数据

在 SQL 支持的所有数据修改中，DELETE 算是最简单的一种了。DELETE 命令的语法为：

```
DELETE FROM <table name>
[WHERE <search condtion>]
```

这里 WHERE 子句也和 SELECT 中的 WHERE 子句相同，只有满足 WHERE 条件的数据行才会被删除。

例如，要删除 ContactType 表中 ContactTypeID 为 21 的数据，则对应的 SQL 语句为：

```
DELETE FROM Person.ContactType
WHERE ContactTypeID=21
```

DELETE 命令也可以像 UPDATE 语句一样，支持使用 FROM 子句来实现多个表的联接删除。例如，要删除 ContactType 表中 Name 字段与 AddressType 表 Name 字段相同的数据行，对应的 SQL 脚本如代码 2.9 所示。

代码 2.9　在 DELETE 中使用 FROM 子句

```
DELETE FROM Person.ContactType
FROM Person.ContactType ct
INNER JOIN Person.AddressType at --通过内联接后再删除
ON ct.Name=at.Name
```

另外，DELETE 也支持 TOP 命令指定删除的行数。TOP 命令的使用方法与 UPDATE 中的 TOP 命令相同。如需要删除 ContactType 表中插入的前 50 行数据，则 SQL 脚本为代码 2.10 所示。

代码 2.10　使用带 TOP 的 DELETE 语句

```
DELETE TOP(50) --只删除匹配的前 50 条数据
FROM Person.ContactType
WHERE ContactTypeID>20
```

在删除行时可能会碰到拒绝删除的情况发生。如果执行：

```
DELETE FROM Person.ContactType
```

系统将抛出异常：

```
消息 547，级别 16，状态 0，第 1 行
DELETE 语句与 REFERENCE 约束"FK_BusinessEntityContact_ContactType_ContactTypeID"冲突。该冲突发生于数据库"AdventureWorks2012"，表"Person.BusinessEntityContact", column 'ContactTypeID'。
语句已终止。
```

这是由于 ContactType 表与其他表建立了外键约束的缘故。在第 3 章将会讲到约束的相关问题。要删除表中的数据必须先删除对应外键约束表中的数据，或者删除约束。

2.3　联 接 查 询

在关系数据库中经常要对多个表进行联合查询并返回一个结果集。在 FROM 子句中可以对表进行重命名，以及对多个表进行联接查询。联接分为内联接、外联接、完全联接和交叉联接。

2.3.1　内联接（INNER JOIN）

内联接是使用比较运算符比较要联接列中的值的联接，是最常用的联接。内联接的语法为：

```
INNER JOIN <table name> ON <join condition>
```

如代码 2.11 所示，a 表联接 sp 表，查询 sp 的 StateProvinceID 与 a 的 StateProvinceID 相等时 a 表 sp 的结果。若某一表中的某行 StateProvinceID 数据为 NULL，则该行数据将不会出现在结果中。

代码 2.11　内联接

```
SELECT a.City,sp.Name
FROM Person.Address a
INNER JOIN Person.StateProvince sp  --内联接
ON sp.StateProvinceID = a.StateProvinceID
```

除了使用 INNER JOIN 进行内联接外，还可以在 WHERE 条件中指定内联接。在 WHERE 子句中使用的内联接被称为旧式内联接，现在已经不推荐使用。例如，要实现与代码 2.11 一样的功能，可以使用 WHERE 子句的内联接代码，如代码 2.12 所示。

代码 2.12　使用 WHERE 子句的内联接

```
SELECT a.City,sp.Name
FROM Person.Address a,Person.StateProvince sp
WHERE sp.StateProvinceID = a.StateProvinceID
--等同于 INNER JOIN，但不推荐使用这样形式
```

INNER JOIN 与接下来要介绍的其他 JOIN 不同的是，INNER JOIN 是排他联接。也就是说，INNER JOIN 联接排除只在一张表中有值而在另一张表中没有值的所有记录。

注意： INNER JOIN 操作是默认的 JOIN 类型，所以在实际查询中可以省略 INNER 关键字。但是省略 INNER 关键字容易使人混淆，所以笔者建议在使用中还是将 INNER 关键字加上为好。

内联接不仅仅只用于两个表的联接，还可以将联接后的结果再与第 3 个和第 4 个表进行联接。如需要将 Address 和 StateProvince，以及 CountryRegion 进行内联接，其 SQL 脚本如代码 2.13 所示。

代码 2.13　多个表的内联接

```
SELECT a.City,sp.Name,cr.Name
FROM Person.Address a
INNER JOIN Person.StateProvince sp --a 表与 sp 表进行内联接
ON sp.StateProvinceID = a.StateProvinceID
INNER JOIN Person.CountryRegion cr --同时再与 cr 表做内联接
ON cr.CountryRegionCode = sp.CountryRegionCode
```

2.3.2　外联接（OUTER JOIN）

外联接分为左外联接（left outer join）和右外联接（right outer join）。

LEFT OUTER JOIN 运算符返回满足第一个（顶端）输入与第二个（底端）输入联接的每一行。它还返回任何在第二个输入中没有匹配行的第一个输入中的行，第二个输入中的非匹配行作为空值返回。相反，RIGHT OUTER JOIN 运算符返回满足第二个（底端）输入与第一个（顶端）输入的每个匹配行的联接的每行。此外，它还返回第二个输入中在第一个输入中没有匹配行的任何行，即与 NULL 联接。

在 OUTER JOIN 中，无论使用左外联接还是右外联接，关键字 OUTER 是可以省略的，正如我们在中文中也习惯与使用左联接和右联接来简称这两种联接一样。省略后，左联接使用 LEFT JOIN 关键字，右联接使用 RIGHT JOIN 关键字。

如代码 2.14 所示，该代码查询 AdventureWorks 公司的所有员工的登录号和部门编号。有兴趣的读者可以将部门表中的数据去掉几条，看看查询结果的变化。

代码 2.14　左联接

```
SELECT e.LoginID,m.DepartmentID
FROM HumanResources.Employee e
LEFT JOIN HumanResources.EmployeeDepartmentHistory m --左联接
ON e.BusinessEntityID=m.BusinessEntityID;
```

在这里，出现在 JOIN 关键字前的表称为左表，出现在 JOIN 关键字后面的表称为右表。如果将左表和右表互换，同时外联接的方向也改变，那么查询出来的结果是一样的。比如将代码 2.14 中的左联接写成右联接就如代码 2.15 所示。

代码 2.15　右联接

```
SELECT e.LoginID,m.DepartmentID
FROM HumanResources.Employee e
RIGHT JOIN HumanResources.EmployeeDepartmentHistory m --右联接
ON e.BusinessEntityID=m.BusinessEntityID;
```

使用外联接的包含特性可以用于查找表中的不匹配记录。在建立了表之间的外键约束情况下一般是不会出现不匹配记录的；但是如果没有建立外键约束，若需要找出不匹配的记录，这时就可以使用外联接。

例如，有班级表 Class 记录了班号 ClassID 和班级名 ClassName，另外有学生表 Student，记录了学号 StudentID、学生姓名 StudentName 和学生所在的班级 ClassID。现在需要找出不在任何班级（即学生表中的 ClassID 无效）的所有学生。使用外联接查找这种不匹配记录的 SQL 脚本，如代码 2.16 所示。

代码 2.16　外联接查找不匹配记录

```
SELECT s.*
FROM Student s
LEFT JOIN Class c
```

```
ON s.ClassID=c.ClassID
WHERE c.ClassID IS NULL --为匹配班号对应的 Class 必定为 NULL
```

2.3.3　完全联接（FULL JOIN）

完全联接与字面意思相同。它包括联接表中的所有行，而不论这些行的数据是否匹配。

注意：FULL JOIN 在实际应用中基本上不会使用。其用途在于不偏爱任何一侧时考虑数据之间的完整关系。

FULL JOIN 可以认为是 LEFT JOIN 和 RIGHT JOIN 的并集。无论是 LEFT JOIN，还是 RIGHT JOIN 产生的结果集，都不会比 FULL JOIN 产生的结果集记录多。例如执行代码 2.17 将得到员工和经理的完全联接。

代码 2.17　完全联接

```
SELECT e.LoginID,m.DepartmentID
FROM HumanResources.Employee e
FULL JOIN HumanResources.EmployeeDepartmentHistory m --完全联接
ON e.BusinessEntityID=m.BusinessEntityID;
```

2.3.4　交叉联接（CROSS JOIN）

交叉联接将第一个（顶部）输入中的每一行与第二个（底部）输入中的每一行联接在一起。交叉联接不需要 ON 子句，返回两个表的笛卡儿积。如果 a 表有 5 行数据，b 表有 6 行数据，对 a、b 表进行交叉联接后，将返回 5×6=30 行数据。

州和地址类型直接没有对应关系表。若希望查询每个州下面的所有地址类型，则可以对州表和地址类型表使用交叉联接。其脚本如代码 2.18 所示。

代码 2.18　交叉联接

```
SELECT sp.StateProvinceID,sp.Name ,at.AddressTypeID,at.Name AS Address-
TypeName
FROM Person.StateProvince sp
CROSS JOIN person.AddressType at --交叉联接
```

说明：若需要对两个以上的表进行多表联接，用户不必关心联接的顺序，无论使用什么联接顺序，SQL Server 都将会对联接进行优化和调整。

交叉联接在实际中也很少使用，主要在进行数据初始化的时候会有一定的用处。例如，在角色权限初始化的时候，希望将当前系统的所有权限配置给每一个管理员使用，此时就可以使用权限和管理员表进行交叉联接，将联接的结果插入到配置表中。

2.3.5　联接的替代写法

在内联接时讲到使用 WHERE 子句来替代联接的写法。除了 FULL JOIN 以外，其他联

接都可以使用 WHERE 子句的方式来替代。但是 WHERE 子句的写法不是标准的 ANSI 写法，并不建议使用。

在替代写法上，OUTER JOIN 的写法与 INNER JOIN 的替代写法相似。但是，因为没有 LEFT 和 RIGHT 关键字，所以需要特殊的操作符“*=”和“=*”来完成。例如，要使用 WHERE 子句来代替代码 2.14 中 LEFT JOIN 查询员工的部门编号，则其 SQL 脚本如代码 2.19 所示。

代码 2.19　LEFT JOIN 的替代写法

```
SELECT e.LoginID,m.DepartmentID
FROM HumanResources.Employee e,HumanResources.EmployeeDepartmentHistory m
WHERE e.BusinessEntityID *= m.BusinessEntityID;
                                    --SQL2008 以后就已经不支持这种过时的写法
```

在 SQL Server 2000 或更早的版本中，运行代码 2.19 得到的结果与代码 2.14 中的结果完全相同。但是在 SQL Server 2012 中却无法得到这样的结果，系统直接认为“*=”是错误的写法。

```
消息 102，级别 15，状态 1，第 3 行
"*="附近有语法错误。
```

同样地，若运行代码 2.20 中 RIGHT JOIN 的 WHERE 替代查询也会出现同样的错误提示。

代码 2.20　RIGHT JOIN 的替代写法

```
SELECT e.LoginID,m.DepartmentID
FROM HumanResources.Employee e,HumanResources.EmployeeDepartmentHistory m
WHERE e.BusinessEntityID =* m.BusinessEntityID;
                              --SQL2008 以后就已经不支持这种过时的写法
```

对于 CROSS JOIN 来说替代写法将十分容易。要使用旧语法创建一个 CROSS JOIN，只需要将两个表放在一起即可，不需要做任何事情。例如，将代码 2.18 中的 CROSS JOIN 替代为旧语法的脚本，如代码 2.21 所示。

代码 2.21　CROSS JOIN 的替代写法

```
SELECT sp.StateProvinceID,sp.Name ,
at.AddressTypeID,at.Name AS AddressTypeName
FROM Person.StateProvince sp , person.AddressType at
```

运行该代码后将得到与代码 2.18 相同的结果。

2.3.6　联合（UNION）

联合操作符是一种用于两个或多个查询产生一个结果集的特殊操作符。UNION 相当于是将两个结果集加起来，将一个结果集添加到另一个结果集的末尾。JOIN 可以认为是将表进行水平组合，而 UNION 则是将表进行垂直组合。

UNION 负责将两个或更多查询的结果合并为单个结果集，该结果集包含联合查询中的所有查询的全部行。UNION 运算不同于使用联接合并两个表中的列的运算，下面列出了使用 UNION 合并两个查询结果集的基本规则。

- 所有查询中的列数和列的顺序必须相同。
- 数据类型必须兼容。

UNION 操作符后可以跟 ALL 选项，表示将全部行并入结果中，其中包括重复行。如果未指定该参数，则删除重复行。例如，要将 AddressType 表中的数据与 ContactType 表中的数据进行联合，则对应脚本如代码 2.22 所示。

代码 2.22　UNION 的使用

```
SELECT at.AddressTypeID,at.Name,at.ModifiedDate
FROM Person.AddressType at
UNION
SELECT ct.ContactTypeID,ct.Name,ct.ModifiedDate
FROM Person.ContactType ct
```

运行后的结果中将出现 AddressTypeID 列、Name 列和 ModifiedDate 列，而不会出现 ContactTypeID 列。这是因为 UNION 联合两个表时，列名使用的是第一个表的列名。另外还可以注意到在前两行将出现两个 AddressTypeID 为 1 的列。也就是说，使用 UNION 后并不是一般认为的将一个表直接跟在另一个表的末尾，而是对新的结果集进行了重新排序。

若运行代码 2.23 使用 UNION ALL 的联合，由于 AddressType 和 ContactType 中并没有重复的数据，所以系统将返回相同的结果数，只是结果中的顺序不同。UNION ALL 才是真正地将一个结果集添加到另一个结果集的末尾，而未做任何处理。

代码 2.23　使用 UNION ALL

```
SELECT at.AddressTypeID,at.Name,at.ModifiedDate
FROM Person.AddressType at
UNION ALL
SELECT ct.ContactTypeID,ct.Name,ct.ModifiedDate
FROM Person.ContactType ct
```

2.4　SQL 数据类型

在 SQL Server 2012 中，每个列、局部变量、表达式和参数都具有一个相关的数据类型属性，用于指定对象可保存的数据类型有整数数据、字符数据、货币数据、日期和时间数据、二进制串和其他类型。按照存储方式的不同，text、ntext、image、varchar(max)、nvarchar(max)、varbinary(max)和 xml 这几种数据类型被分为大对象数据类型（BLOB）。

另外，SQL Server 还支持用户自定义数据类型和 CLR 数据类型。本书中将进行专门的讲解，在此不做讨论。按照数据类型表示的数据来分，数据类型被分为 7 类，具体分类如表 2.2 所示。

表 2.2　数据类型分类

类 型 分 类	SQL 类型
精确数字	bigint decimal int numeric smallint money tinyint smallmoney bit
近似数字	float real
字符串	char text varchar
Unicode 字符串	nchar ntext nvarchar
二进制串	binary image varbinary
日期和时间	datetime smalldatetime date time datetime2 datetimeoffset
其他数据类型	cursor timestamp sql_variant uniqueidentifier table xml CLR 自定义类型

2.4.1　精确数字类型

精确数字类型是指能够精确地表示某一数值的数据类型，主要有以下几种类型。

- Bit 类型相当于其他编程语言中的布尔类型。该类型只有 0 和 1 两种值。虽然 Bit 类型相当于编程语言中的布尔值，但是在 SQL Server 中并不能将其作为布尔值进行逻辑判断。
- Tinyint 类型占用 1B 的存储空间，用于表示 0～255 的整数。
- Smallint 类型占用 2B 的存储空间。用于表示–2^15（–32 768）～2^15–1（32 767）之间的整数。
- Int 类型占用 4B 的存储空间，用于表示–2^31（–2 147 483 648）～2^31–1（2 147 483 647）之间的整数。该类型在整形数据中最为常用。
- Bigint 类型占用 8B 的存储空间，用于表示–2^63（–9 223 372 036 854 775 808）～2^63–1（9 223 372 036 854 775 807）的整数。
- Smallmoney 类型主要用于存储货币值，占用 4B 的存储空间，用于表示–214 748.3648～214 748.3647 之间的数。该类型精确到万分位。
- Money 类型也是用于存储货币值，其精度与 smallmoney 相同，不过它占用了 8B 的存储空间，用于表示–9 223 372 036 854 775 808～9 223 372 036 854 775 807 之间的数据。
- Decimal 类型和 Numeric 类型在 SQL Server 中是等价的，用于表示指定精度和小数位的数据类型。它们的定义式为 decimal[(p[, s])]和 numeric[(p[, s])]。其中 p 表示最多可以存储的十进制数字的总位数，包括小数点左边和右边的位数。该精度必须是从 1 到最大精度 38 之间的值，默认精度为 18。s 表示小数点右边可以存储的十进制数字的最大位数。小数位数必须是 0～p 之间的值，仅在指定精度后才可以指定小数位数。默认的小数位数为 0，因此，0≤s≤p。最大存储大小基于精度而变化。

2.4.2　近似数字类型

float 类型和 real 类型用于表示浮点数值数据的大致数值数据类型。浮点数据为近似值，因此，并非数据类型范围内的所有值都能精确地表示。

float 类型的定义式为 float[(n)]。其中 n 为用于存储 float 数值尾数的位数，以科学记数

法表示，因此可以确定精度和存储大小。如果指定了 n，则它必须是介于 1～53 之间的某个值。n 的默认值为 53。当 n 小于等于 24 时，float 类型占用 4B 的存储空间；当 n 大于 24 时，float 将占用 8B 的存储空间。

说明： real 类型就相当于 float(24)。

2.4.3　字符串类型

在 SQL Server 中字符串类型被分为 3 种：固定长度字符串、可变长度字符串和大对象字符串，对应的数据类型就是 char、varchar 和 text。SQL Server 中使用“'”来表示字符串，字符串内容都必须放在“'”符号之中。比如：

```
'Test String'
```

固定长度字符串类型的定义式为 char [(n)]，其用于存储非 Unicode 字符数据，长度为 n 个字符。n 的取值范围为 1～8000，存储大小是 nB。固定长度意味着该数据类型的实际长度是固定不变的。当其中的数据并没有达到定义的长度值时，系统将在后面填充空数据以使数据长度达到定义值。

可变长度字符串类型的定义式为 varchar [(n)]，其用于可变长度的存储非 Unicode 字符数据。n 的取值范围为 1～8000。另外，在 SQL Server 2008 中，n 可以写为 max 表示最大存储大小是 2^31-1B。存储大小是输入数据的实际长度加 2B，所输入数据的长度可以为 0 个字符。

说明： char 和 varchar 中的 n 若没有指定，则系统默认为一个字符。

text 数据类型与 varchar(max)相同，用于表示 8000 个字符以上的字符串。Text 类型已经是过时的数据类型，微软在将来的 SQL Server 版本中考虑将取消这种数据类型。

对于这 3 种字符串类型其选择使用的原则是：

- 如果列数据项的大小一致，则使用 char。
- 如果列数据项的大小差异相当大，则使用 varchar。
- 如果列数据项大小相差很大，而且大小可能超过 8000B，则使用 varchar(max)。

注意： char 和 varchar 类型也可以存储 Unicode 字符数据。不过由于 Unicode 字符使用 2B 表示一个字符，所以存储的字符串长度将减半。也就是说，char(10)的数据类型只能存储 5 个汉字。

2.4.4　Unicode 字符串类型

SQL Server 中使用 nchar、nvarchar 和 text 来存储 Unicode 字符串。它的定义式和使用方法与非 Unicode 字符串基本相同，只是 Unicode 字符使用 2B 来表示一个字符，所以 nchar 和 nvarchar 接受的最大长度是 4000 个字符。另外，Unicode 字符串在申明时需要在“'”符号前加 N。例如：

```
N'测试字符串'
```

Ntext 和 text 都是作为一种过时的时间类型即将被淘汰，建议使用 nvarchar(max)类型。

2.4.5　二进制串类型

SQL Server 中除了可以存储文本信息外，还可以存储二进制信息。二进制串的数据类型和字符串相同，分为固定长度、可变长度和大对象二进制串 3 种。对应的数据类型是 binary、varbinary 和 image。SQL Server 中使用整数来表示二进制数据内容，更普遍的是使用十六进制数来表示。例如：

```
0x020000005607F91E159702D67280375D17C80D3DA3044E23
```

前面的 0x 就表明了后面的数据是十六进制数据。二进制串的定义和使用与字符串基本相同，binary 和 varbinary 的最大长度也是 8000B。Image 作为一种过时的数据类型即将被淘汰，建议使用 varbinary(max)类型。

2.4.6　日期和时间类型

SQL Server 开始就支持了两种与日期和时间相关的数据类型，即 datetime 和 smalldatetime。在 SQL Server 2008 中又添加了 4 种与日期和时间相关的数据类型，即 DATETIME2、DATE、TIME 和 DATETIMEOFFSET。在目前的 SQL Server 2012 中并没有再次补充相应的类型。

Datetime 数据类型需要 8B 的存储空间，其中前 4 个字节用 1900 年 1 月 1 日以后的天数表示日期。后 4 个字节表示一天中的时间，从 00：00：00 开始，以 3.333 毫秒为单位。Datetime 支持的日期范围是从 1753-1-1～9999-12-31。Datetime 类型最早只支持 1753 年是因为历史原因造成的。以前在西方使用两种历法：儒略历和格里历。

这两种历法之间相差数天。也就是说，使用儒略历的国家指的 1752-1-1 与使用格里历的国家指的 1752-1-1 并不是同一天。为此英国在 1752 年作了转换（在这一年，1752-9-2 的下一天是 1752-9-14），随后土耳其、瑞典等国也采用不同的方式对历法进行了转换。由于这个原因，1753 年前的日期在未指定历法的情况下，并不确定是历史上具体的哪一天。所以 Sybase SQL Server（Microsoft SQL Server 的前身）决定不允许使用 1753 年前的日期。

Smalldatetime 数据类型需要 4B 的存储空间，其中前两个字节表示 1900-1-1 年以后的所有天数。另外两个字节表示 00：00 以后的分钟数。该类型支持的范围从 1900-1-1～2079-6-6。在定义某一具体的日期时间时可以使用字符串按照日期时间的格式进行定义。系统将会自动把该字符串转换为日期时间类型。例如：

```
'2007-12-31 23:59:59.9'
```

DATETIME2 数据类型视做现有 DATETIME 类型的扩展，其数据范围更大，默认的小数精度更高，并具有可选的用户定义的精度。DATETIME2 表示的日期范围从 0001-01-01～9999-12-31，而表示的时间从 00:00:00～23:59:59.9999999，最高精度可达 100 纳秒。

使用 DATETIME2 数据类型时，可以使用不同的长度字符存储和显示日期，从 19（YYYY-MM-DD hh:mm:ss）到 27（YYYY-MM-DD hh:mm:ss.0000000），占用的数据空

间也从 6～8 个字节不等。在定义 DATETIME2 数据类型时可以指定保留的时间精度从 0～7，如果未指定则使用默认的 7 位时间小数的精度。

DATETIME2(3)格式等同于 SQL Server 中使用的 DATETIME 格式，但是使用 DATETIME2(3)却可以获得更高的精度、更宽的日期范围，同时还更节约存储空间。前面介绍了 DATETIME 格式只能精确到 3.33 毫秒，而 DATETIME2(3)支持精确度到 1 毫秒。DATETIME2(3)的日期范围是从 0001-01-01～9999-12-31，比 DATETIME 的日期范围要广。另外，DATETIME 数据类型需要占用 8 个字节的存储空间，而 DATETIME2(3)只占用 7 个字节。如表 2.3 所示为 DATETIME2 数据类型的不同精度所对应的列长度和小数。

表 2.3　DATETIME2 数据类型精度

指定的小数位数	结果精度	小数位数	列长度（以字节为单位）	小数，秒精度
datetime2	27	7	8	7
datetime2(0)	19	0	6	0～2
datetime2(1)	21	1	6	0～2
datetime2(2)	22	2	6	0～2
datetime2(3)	23	3	7	3～4
datetime2(4)	24	4	7	3～4
datetime2(5)	25	5	8	5～7
datetime2(6)	26	6	8	5～7
datetime2(7)	27	7	8	5～7

例如，使用不同精度的 DATETIME2 数据类型来存储一个日期时间数据，输出不同精度日期时间数据的脚本，如代码 2.24 所示。

代码 2.24　不同精度的 DATETIME2 变量

```
--定义不同精度的 DATETIME2 数据类型的变量
DECLARE @D0 datetime2(0) = '1984-12-29 13:14:15.1234567';
DECLARE @D1 datetime2(1) = '1984-12-29 13:14:15.1234567';
DECLARE @D2 datetime2(2) = '1984-12-29 13:14:15.1234567';
DECLARE @D3 datetime2(3) = '1984-12-29 13:14:15.1234567';
DECLARE @D4 datetime2(4) = '1984-12-29 13:14:15.1234567';
DECLARE @D5 datetime2(5) = '1984-12-29 13:14:15.1234567';
DECLARE @D6 datetime2(6) = '1984-12-29 13:14:15.1234567';
DECLARE @D7 datetime2(7) = '1984-12-29 13:14:15.1234567';
--输出各个精度变量的值
PRINT @D0;
PRINT @D1;
PRINT @D2;
PRINT @D3;
PRINT @D4;
PRINT @D5;
PRINT @D6;
PRINT @D7;
```

输出结果为：

```
1984-12-29 13:14:15
```

```
1984-12-29 13:14:15.1
1984-12-29 13:14:15.12
1984-12-29 13:14:15.123
1984-12-29 13:14:15.1235
1984-12-29 13:14:15.12346
1984-12-29 13:14:15.123457
1984-12-29 13:14:15.1234567
```

DATE 数据类型用于定义表示一个日期数据类型。DATE 数据类型表示的日期范围从 0001-01-01～9999-12-31，占用 3 个字节的存储空间。DATE 数据类型用于存储生日、入职日期等之类只需要日期表示的数据。

在将 DATE 转换成字符串时，默认使用 YYYY-MM-DD 的格式表示，占用 10 个字节。在将字符串转换成 DATE 类型时则不一定要符合该格式，只要符合数字日期格式，都可以被系统识别并转换成 DATE 类型。例如，定义一个 DATE 类型，然后将该类型输出的脚本，如代码 2.25 所示。

代码 2.25　使用 DATE 类型

```
DECLARE @d date
SET @d='1984/2/29'
PRINT @d
```

输出结果为：

```
1984-02-29
```

注意：一个比截止年份的后两位数字小，或者与其相等的两位数年份与该截止年份处于同一个世纪。而一个比截止年份的后两位数字大的两位数年份所处的世纪为该截止年份所处世纪的上一个世纪。例如，“两位数年份截止”为默认值 2049，则两位数年份 49 被解释为 2049 年。而两位数年份 50 被解释为 1950 年，所以并不建议使用两位数年份。

TIME 数据类型用于表示一天中的某个时间。该时间使用 24 小时表示，而且与时区无关。TIME 数据类型支持从 0～7 不同的时间精度，最高精度为 100 纳秒，占用磁盘空间 3～5 个字节。默认情况下 TIME 类型具有 7 位的精度。表 4.2 列出了不同的精度情况下 TIME 类型占用的字节数和秒的精度。

表 2.4　TIME类型精度

指定的小数位数	结果精度	小数位数	列长度（以字节为单位）	小数，秒精度
time	16	7	5	7
time(0)	8	0	3	0～2
time(1)	10	1	3	0～2
time(2)	11	2	3	0～2
time(3)	12	3	4	3～4
time(4)	13	4	4	3～4
time(5)	14	5	5	5～7
time(6)	15	6	5	5～7
time(7)	16	7	5	5～7

TIME 转换成字符串后最小长度为 8 位，最大长度为 16 位。例如，对于不同精度的 TIME 数据，输出其字符串的脚本，如代码 2.26 所示。

代码 2.26　TIME 不同精度

```
--定义不同精度的时间类型变量
DECLARE @t0 time(0) = '13:14:15.1234567';
DECLARE @t1 time(1) = '13:14:15.1234567';
DECLARE @t2 time(2) = '13:14:15.1234567';
DECLARE @t3 time(3) = '13:14:15.1234567';
DECLARE @t4 time(4) = '13:14:15.1234567';
DECLARE @t5 time(5) = '13:14:15.1234567';
DECLARE @t6 time(6) = '13:14:15.1234567';
DECLARE @t7 time(7) = '13:14:15.1234567';
--打印时间变量
PRINT @t0;
PRINT @t1;
PRINT @t2;
PRINT @t3;
PRINT @t4;
PRINT @t5;
PRINT @t6;
PRINT @t7;
```

输出结果为：

```
13:14:15
13:14:15.1
13:14:15.12
13:14:15.123
13:14:15.1235
13:14:15.12346
13:14:15.123457
13:14:15.1234567
```

从输出结果中可以看出，TIME 数据类型与 DATETIME2 数据类型类似，只是 TIME 类型只存储了时间部分，从而减少了存储空间的大小。

DATETIMEOFFSET 数据类型与 DATETIME2 数据类型相似，表示日期和时间的组合，其最高精度也是 100 纳秒。不过 DATETIMEOFFSET 是识别时区的，时区的范围从-14:00～+14:00，这个值是增加或者减去 UTC 以获取本地时间。由于需要存储时区，所以相对于 DATETIME2，在同样的精度情况下，需要增加额外的 2 个字节。

DATETIMEOFFSET 转换为字符串时，最低精度情况下需要 26 个字节的字符串表示，形如：YYYY-MM-DD hh:mm:ss {+|-}hh:mm，最高精度情况下需要 34 个字节来表示，形如：YYYY-MM-DD hh:mm:ss.nnnnnnn {+|-}hh:mm。表 2.5 列出了不同精度情况下 DATETIMEOFFSET 数据类型的列长度。

表 2.5　DATETIMEOFFSET数据类型的精度

指定的小数位数	结果精度	小数位数	列长度（以字节为单位）	秒的小数部分精度
datetimeoffset	34	7	10	7
datetimeoffset(0)	26	0	8	0～2
datetimeoffset(1)	28	1	8	0～2
datetimeoffset(2)	29	2	8	0～2

续表

指定的小数位数	结果精度	小数位数	列长度（以字节为单位）	秒的小数部分精度
datetimeoffset(3)	30	3	9	3～4
datetimeoffset(4)	31	4	9	3～4
datetimeoffset(5)	32	5	10	5～7
datetimeoffset(6)	33	6	10	5～7
datetimeoffset(7)	34	7	10	5～7

例如，定义一个具有 5 位小数精度的 DATETIMEOFFSET 数据类型，时区为+08:00，然后输出其标准格式结果的脚本，如代码 2.27 所示。

代码 2.27　DATETIMEOFFSET 的使用

```
DECLARE @d DATETIMEOFFSET(5) = '1984-12-29 13:14:15.1234567 +08:00'
PRINT @d
```

输出结果为：

```
1984-12-29 13:14:15.12346 +08:00
```

通过使用 DATETIME2、DATE、TIME 和 DATETIMEOFFSET 数据类型，可以获得更大的日期范围、更高的数据精度，同时也更节约存储空间。这些新日期和时间类型扩展应用程序使用的范围，存储日期使用正确的格式而不需要写大量的自定义代码。

2.4.7　其他数据类型

为了实现某些特殊的功能，SQL Server 还提供了几种其他的数据类型。

- Uniqueidentifier 数据类型：用于表示一个全球唯一标识符（GUID），该类型使用 16B 的存储空间。其内容形如 6F9619FF-8B86-D011-B42D-00C04FC964FF，用户可以通过使用 NEWID 函数生成或者将 GUID 形式的字符串转换为该时间类型。由于 Uniqueidentifier 类型的值是唯一的，所以一般作为表的主键使用。
- timestamp 数据类型：用于公开数据库中自动生成的唯一二进制数字的数据类型。timestamp 通常用于给表行加版本戳的机制。存储大小为 8B。每个数据库都有一个计数器，当对数据库中包含 timestamp 列的表执行插入或更新操作时，该计数器值就会增加。该计数器是数据库时间戳，可以跟踪数据库内的相对时间，而不是时钟相关联的实际时间。一个表只能有一个 timestamp 列。每次修改或插入包含 timestamp 列的行时，就会在 timestamp 列中插入增量数据库时间戳值。
- sql_variant 数据类型：作为一种通用数据类型可以存放不同数据类型的数据。但是 varchar(max)、varbinary(max)、nvarchar(max)、xml、text、ntext、image、timestamp、sql_variant 和用户定义类型的数据不能存放。sql_variant 的最大长度可以是 8016B，这其中包括基类型信息和基类型值。实际基类型值的最大长度是 8000B。
- cursor 类型、table 类型以及 xml 数据类型：这些类型将会在本书中做专门的讲解，在此不做介绍。

2.5　SQL 变量

SQL 变量是 Transact-SQL 批处理和脚本中可以保存数据值的对象。声明或定义此变量后，批处理中的一个语句可以将此变量设置为一个值，该批处理中后面的一个语句可以从此变量中获取这个值。批处理和脚本中的变量通常用于以下几个方面。

- 作为计数器计算循环执行的次数或控制循环执行的次数。
- 保存数据值以供控制流语句测试。
- 保存存储过程返回代码要返回的数据值或函数返回值。

SQL 变量的名称必须以@符号开头，DECLARE 语句用于声明并初始化 SQL 变量，其语法形式为：

```
DECLARE @userName varchar(50)
```

其中，@userName 是声明的变量名，varchar(50)是该变量的数据类型。

DECLARE 语句可以一次声明多个变量。比如：

```
DECLARE @userName varchar(50),@age int,@sex bit
```

第一次声明变量时，其值设置为 NULL。若要为变量赋值，可使用 SET 语句，这是为变量赋值的首选方法。比如：

```
DECLARE @userName varchar(50)
SET @userName='Bill'
```

也可以通过 SELECT 语句的选择列表中当前所引用的值为变量赋值。例如代码 2.28 所示，定义了一个字符串变量@name 和一个日期时间变量@date，然后使用 SELECT 命令对这两个变量进行赋值。

代码 2.28　变量的使用

```
DECLARE @name varchar(50),@date datetime --定义变量
SELECT @name = Name,@date = ModifiedDate --变量赋值
FROM Person.AddressType
WHERE AddressTypeID=1
```

在为变量赋值后，接下来的 SQL 语句便可以使用该变量中的值了。变量的值可以作为参数传给其他函数或存储过程，也可以作为 SQL 语句中的一部分参与其他操作。例如代码 2.29 所示，在对变量赋值后便可输出变量或将变量作为参数。

代码 2.29　输出变量和使用变量作参数

```
DECLARE @name varchar(50),@date datetime
SELECT @name = Name,@date = ModifiedDate
FROM Person.AddressType
WHERE AddressTypeID=1
PRINT @name --打印出变量@name
```

```
SELECT YEAR(@date) --在 YEAR 函数中使用变量@date 作为参数
```

2.6 操　作　符

在 SQL Server 的操作符中最常用的便是运算符。运算符按照功能的不同，可以分为下面几种。

- 算术运算符：+、–、*、/、%（取模）。
- 位运算算运算符：&（与）、|（或）、^（异或）、~（求反）。
- 逻辑运算符：AND、OR、NOT。
- 比较运算符：<、>、=、>=（!<）、<=（!>）、<>（!=）。
- 字符串运算符：+实现字符串之间的连接操作。如'abc'+'123'的结果为'abc123'。

这些运算符的意义与编程语言中的运算符相同，笔者在此就不对每个运算符的作用和使用方法做解释了。以上提到的操作符都是针对两个对象的操作。为了便于对集合进行操作，SQL Server 还提供了大量的集合操作符。

（1）[NOT] IN 判断一个对象是否在另一个集合中。这两个操作符返回一个布尔值，可以用于条件判断和 WHERE 子句中。其使用方法如代码 2.30 所示。

代码 2.30　IN 操作符的使用

```
IF(2 IN (1,2,3,4))--在条件语句中使用
PRINT 'ok'
-------------------------------
SELECT a.*
FROM Person.Address a
WHERE a.StateProvinceID NOT IN (
    SELECT sp.StateProvinceID
    FROM Person.StateProvince sp
    WHERE sp.CountryRegionCode='US'
    )--在 WHERE 子句使用
```

（2）ANY 和 SOME 需要比较运算符一起使用，主要用对象与一个集合中的对象逐一比较，若任一比较结果为真，则返回 True；否则返回 False。另外，在 SQL Server 中 ANY 和 SOME 的作用是相同的。其使用方法如代码 2.31 所示。

代码 2.31　ANY、SOME 操作符的使用

```
IF(5>SOME(SELECT AddressTypeID
FROM Person.AddressType)
)--因为 5>1 所以该表达式为真
PRINT 'ok'
-----------------------------
IF(7<=ANY(SELECT AddressTypeID
FROM Person.AddressType)
)--因为 7>1,2,3,4,5,6 所以该表达式为假
PRINT 'ok'
```

（3）ALL 与 SOME 相似，需要与比较运算符一起使用。但 ALL 操作符的作用是用对象与集合中的对象逐一比较。若比较结果全部为真，则返回 True；否则返回 False。其使用方法如代码 2.32 所示。

代码 2.32　ALL 操作符的使用

```
IF(5>=ALL(SELECT AddressTypeID
    FROM Person.AddressType)
)--因为 5<6 所以该表达式为假
PRINT 'ok'
```

（4）BETWEEN 操作符除用于数字区间外，还可以用于时间区间和字符串区间。其使用方法如代码 2.33 所示。

代码 2.33　BETWEEN 操作符的使用

```
---------以下是 between 的字符串比较-------
SELECT *
FROM Person.AddressType
WHERE [Name] NOT BETWEEN 'A' AND 'C'
---------以下是 between 的数字比较---------
IF(12 BETWEEN 1 AND 20)
    PRINT 'between 1 and 20'
---------以下是 between 的日期比较---------
SELECT *
FROM HumanResources.Employee
WHERE BirthDate BETWEEN '1980-1-1' AND '2000-12-31'
```

（5）[NOT] EXISTS 操作符用于检查一个子查询。如果子查询的结果包含任何行，则返回 True；反之则返回 False。该操作符除用于查询的 WHERE 子句中外还常用于检查系统中是否存在指定数据库对象。其使用方法如代码 2.34 所示。

代码 2.34　[NOT] EXISTS 操作符的使用

```
SELECT a.*
FROM Person.Address a
WHERE NOT EXISTS( --是否返回行
    SELECT sp.StateProvinceID
    FROM Person.StateProvince sp
    WHERE sp.CountryRegionCode='US' AND
    a.StateProvinceID=sp.StateProvinceID
    )
```

（6）IS [NOT] NULL 操作符是用于判断一个对象是否为空。如果对象为空则返回 True；否则返回 False。其使用方法如代码 2.35 所示。

代码 2.35　IS[NOT] NULL 操作符的使用

```
SELECT *
FROM HumanResources.EmployeeDepartmentHistory
WHERE EndDate IS NOT NULL
```

注意：默认情况下只能用 IS NULL 判断一个对象是否为空，而不能用=NULL 进行判断，但是在当前连接运行“SET ANSI_NULLS OFF”后可以是=NULL 来判断空值。

（7）[NOT] LIKE 操作符用于确定特定字符串是否与指定模式相匹配。模式可以包含常规字符和通配符。模式匹配过程中，常规字符必须与字符串中指定的字符完全匹配，但是通配符可以与字符串的任意部分相匹配。与使用“=”和“!=”字符串比较运算符相比，使用通配符可使 LIKE 运算符更加灵活。LIKE 操作符最常用的功能就是使用“%”来进行模糊搜索。“%”用于表示 0 个或多个任意字符，如要搜索部门名中含有“Pro”这 3 个字母的所有部门。其 SQL 语句如代码 2.36 所示。

代码 2.36　LIKE 操作符的使用

```
SELECT *
FROM HumanResources.Department
WHERE Name LIKE '%Pro%' --模糊匹配
```

关于字符串的模式匹配除了“%”外还有其他模式，读者可以自行翻阅联机丛书。表 2.6 列出了 SQL Server 中的所有操作符及其示例以便读者查看。

表 2.6　操作符说明

操　作　符	操　　作	示　　例
+ - * / %	数学运算	11+2 5*25
& | ^ ~	位运算	2&3
AND OR NOT	逻辑运算	@id>1 AND @s=10
> < = >= <= != <> !> !<	比较运算	@id!=10
+	字符串运算	'abc'+'defg'
IN	在集合中	@id IN (1,2,3,4)
ANY SOME	集合中的任何值比较	@id>ANY(1,3,5,7)
ALL	集合中的所有值比较	@id>ALL(1,3,5,7)
BETWEEN	在区间	@id BETWEEN 1 AND 10
EXISTS	存在查询结果	EXISTS(SELECT * FROM T1)
IS NULL	为空	@id IS NULL
LIKE	模式匹配	WHERE UName LIKE 'Bill%'

2.7　流 程 控 制

T-SQL 语言也像其他程序设计语言一样具有顺序语句、条件语句和循环语句等流程控制，使用流程控制可以在 T-SQL 中实现复杂的逻辑。

2.7.1　批处理

批处理（简称批）是同时从应用程序发送到 SQL Server 2012 并得以执行的一组单条

或多条 Transact-SQL 语句。SQL Server 将批处理的语句编译为单个可执行单元，称为执行计划。执行计划中的语句每次执行一条。

每个不同的批处理之间使用 GO 进行分隔。GO 并不是 T-SQL 语句，其作用只是告诉 SQL Server 实用工具（例如 SSMS）将当前 GO 命令之前，上一个 GO 命令之后的所有 SQL 语句作为一个批处理发送到数据库引擎。GO 命令不能和其他 T-SQL 语句在同一行中，但是 GO 命令行中可以包含注释，如代码 2.37 所示。其中，第 5 行将报错："出现脚本错误。分析 GO 时遇到错误语法。"

代码 2.37　GO 命令

```
USE AdventureWorks2012;
GO
SELECT *
FROM Person.AddressType
GO SELECT * --此处报错
FROM Person.ContactType
```

另外局部（用户定义）变量的作用域限制在一个批处理中，不可在 GO 命令后引用，如代码 2.38 所示，运行将报错："必须声明标量变量 "@MyMsg""。

代码 2.38　GO 与局部变量

```
USE AdventureWorks2012;
GO
DECLARE @MyMsg VARCHAR(50)
SELECT @MyMsg = 'Hello, World.'
GO -- 批处理结束@MyMsg 的作用域也结束
PRINT @MyMsg --此处报错
GO
```

注意：如果基于 ODBC 或 OLE DB API 的应用程序试图执行 GO 命令，会收到语法错误。SQL Server 实用工具从不向服务器发送 GO 命令。

SQL Server 实用工具在提交 SQL 脚本到数据库引擎之前先对脚本内容进行分析，根据 GO 命令将脚本分为一个一个的批处理，并分别依次提交到数据库引擎。若用户一次提交了两个批处理，如果第一个批处理编译错误而第二个批处理正确，那么 SQL Server 将抛出异常，不会执行第一个批处理。但是 SQL Server 实用工具并不会因为异常的产生而停止执行，第二个批处理将被提交到数据库引擎并正确执行。

而在一个批处理中如果编译错误，由于一个批处理是作为一个整体进行编译后再运行的，所以整个批处理中的命令将都不会得到执行。但是对于算术溢出或约束冲突之类的运行时错误将会有如下影响：

- 大多数运行时错误将停止执行批处理中当前语句和它之后的语句。
- 某些运行时错误（如违反约束）仅停止执行当前语句，而继续执行批处理中其他所有语句。

在遇到运行时错误的语句之前执行的语句不受影响。唯一例外的情况是，批处理位于事务中并且发生错误导致事务回滚，在这种情况下，所有在运行时错误之前执行的未提交

数据修改都将回滚。

代码 2.39　批处理中的异常

```
CREATE TABLE dbo.t3(a int)
INSERT INTO dbo.t3 VALUES (1)        --此行正确执行
INSERT INTO dbo.t3 VALUES (1,1)      --此行报错，该行及下一行不会被执行
INSERT INTO dbo.t3 VALUES (3)
GO
SELECT * FROM dbo.t3                 --返回 1 行结果
```

代码 2.39 中有两个批处理，在执行时第 3 行发生运行时错误，所以第 3 行和第 4 行将不会执行。在执行了第一个批后，第二个批将接着执行并返回一行结果，该结果便是第二行执行的结果。

2.7.2　语句块

T-SQL 中使用 BEGIN…END 来指定语句块，语句块相当于 C 语言中的{…}，其使用方法也与 C 语言差不多。语句块主要用于 IF 语句和 WHILE 语句后，用以表示该语句块中的语句在该条件下运行。语句块也可以进行嵌套。

语句块与一般编程语言有所不同的就是，在语句块中声明的变量其作用域是在声明后的整个批中，而不仅仅局限于这个语句块中。如代码 2.40 所示，在大多数编程语言中这种写法是错误的。因为@str 的作用域在 END 语句结束，而此处脚本将正常运行。

代码 2.40　语句块中的变量

```
DECLARE @id int
SET @id=1                       --读者可以将值改为 2 再运行一下
IF(@id=1)
BEGIN
    DECLARE @str varchar(10)    --在语句块中定义变量
    SET @str='abc'
END
SELECT '['+@str+']'             --在语句块外使用该变量
```

2.7.3　条件语句

在 T-SQL 中的条件语句包括 IF 和 ELSE，多条 IF 语句可以嵌套使用。IF 和 ELSE 后跟着语句块，但是当语句块中只有一条语句时可以忽略 BEGIN…END，直接跟 SQL 语句。比如：

```
IF(@id>5)
    PRINT @id
```

在大范围的条件判断中，使用 CASE 语句会更简单一些。CASE 语句计算条件列表并返回多个可能结果表达式之一。CASE 具有两种格式：

- 简单 CASE 函数将某个表达式与一组简单表达式进行比较以确定结果。
- CASE 搜索函数计算一组布尔表达式以确定结果。

这两种格式都支持可选的 ELSE 参数。简单 CASE 函数的格式如代码 2.41 所示。

代码 2.41　简单 CASE 格式

```
CASE input expression
    WHEN when expression THEN result expression
   [ ...n ]
    [
   ELSE else result expression
    ]
END
```

简单 CASE 函数就是针对 input_expression 为不同的值时，返回对应的结果。例如代码 2.42 便是简单 CASE 函数的使用。

代码 2.42　简单 CASE 使用

```
SELECT * ,
CASE EmailPromotion
WHEN 0 THEN 'No Email'
WHEN 1 THEN 'Only AdventureWork'
WHEN 2 THEN 'AdventureWork and Selected Email'
ELSE 'Other Email'
END AS EmailPromotion
FROM Person.Person
```

CASE 搜索函数的格式如代码 2.43 所示。

代码 2.43　CASE 搜索函数格式

```
CASE
    WHEN Boolean expression THEN result expression
   [ ...n ]
    [
   ELSE else result expression
    ]
```

该格式没有 input_expression，但是在 WHEN 后必须跟返回布尔值的表达式。如果表达式返回的结果为 True，则最终返回 result_expression 并结束整个 CASE 语句。如果所有 WHEN 表达式返回 False，则返回 ELSE 后的结果。代码 2.44 为 CASE 搜索函数的使用样例。

代码 2.44　CASE 搜索函数的使用

```
SELECT *,CASE
WHEN EmailPromotion=0 THEN 'No Email'
WHEN EmailPromotion=1 THEN 'Only AdventureWork'
WHEN EmailPromotion=2 THEN 'AdventureWork and Selected Email'
ELSE 'Other Email'
END AS EmailPromotion
FROM Person.Person
```

2.7.4　循环语句

为了实现循环，T-SQL 中提供了 WHILE 循环和 GOTO 语句。WHILE 循环的格式如

代码 2.45 所示。

代码 2.45　WHILE 循环格式

```
WHILE Boolean expression
     { sql_statement | statement_block }
     [ BREAK ]
     { sql_statement | statement_block }
     [ CONTINUE ]
     { sql_statement | statement_block }
```

（1）WHILE 后跟返回布尔值的表达式。如果表达式返回 True 则执行该循环，直到表达式返回 False 才结束循环。WHILE 循环可以进行嵌套。

（2）BREAK 导致从最内层的 WHILE 循环中退出，程序将执行出现在 END 关键字（循环结束的标记）后面的任何语句。如果嵌套了两个或多个 WHILE 循环，则内层的 BREAK 将退出到下一个外层循环，程序将首先运行内层循环结束之后的所有语句，然后重新开始下一个外层循环。

（3）CONTINUE 使 WHILE 循环重新开始执行，忽略 CONTINUE 关键字后面的任何语句。代码 2.46 为 WHILE 循环的简单使用实例，此实例实现 0～9 的输出。

代码 2.46　使用 WHILE 循环输出 0～9

```
DECLARE @id int
SET @id=0
WHILE (@id<10)
BEGIN
    PRINT @id
    SET @id=@id+1
END
```

除了 WHILE 语句外 GOTO 语句与 IF 语句一起使用也可以实现循环。GOTO 语句可以实现跳转到某一标记处。在跳转过程中，GOTO 语句与标志语句之间没有语句被执行。GOTO 语句的使用如代码 2.47 所示，该代码的作用是实现数字 0～9 的输出。

代码 2.47　使用 GOTO 语句输出 0～9

```
DECLARE @id int
SET @id=0
lb:--标记
PRINT @id
SET @id=@id+1
IF (@id<10)
    GOTO lb --跳转到标记 lb
```

2.8　函　　数

与一般编程语言一样，函数是 SQL Server 中非常重要的功能之一。SQL Server 提供了大量可用于执行特定操作的内置函数。本节将主要讲解这些内置函数。

2.8.1　函数简介

函数用于实现一定功能的功能块。在 SQL Server 中，函数分为内置函数和用户自定义函数。函数可用于或包括在以下几个方面。

- 使用 SELECT 语句的查询的选择列表中，以返回一个值。
- SELECT 或数据修改（SELECT、INSERT、DELETE 或 UPDATE）语句的 WHERE 子句搜索条件中，以限制符合查询条件的行。
- 视图的搜索条件（WHERE 子句）中，以使视图在运行时与用户或环境动态地保持一致。
- 任意表达式中。
- CHECK 约束或触发器中，以在插入数据时查找指定的值。
- DEFAULT 约束或触发器中，以在 INSERT 语句未指定值的情况下提供一个值。

说明：指定函数时应始终带上括号，即使没有参数也是如此。但是与 DEFAULT 关键字一起使用的 niladic 函数例外。

表 2.7 列出了 SQL Server 函数的类别。

表 2.7　函数的类别

函 数 类 别	说　　明
聚合函数（T-SQL）	执行的操作是将多个值合并为一个值。例如 AVG、COUNT 和 MAX 等
配置函数	是一种标量函数，可返回有关配置设置的信息
加密函数（T-SQL）	支持加密、解密、数字签名和数字签名验证
游标函数	返回有关游标状态的信息
日期和时间函数	对日期和时间进行处理，也可以更改日期和时间的值
数学函数	执行三角、几何和其他数学运算
元数据函数	返回数据库和数据库对象的属性信息
排名函数	SQL 2005 新增，是一种非确定性函数，可以返回分区中每一行的排名值
行集函数（T-SQL）	返回可在 T-SQL 语句中表引用所在位置使用的行集
安全函数	返回有关用户和角色等安全相关的信息
字符串函数	对字符串和二进制进行处理
系统函数	对系统级的各种选项和对象进行操作或报告
系统统计函数（T-SQL）	返回有关 SQL Server 性能的信息
文本和图像函数	可更改 text 和 image 等大对象的值

在 SQL Server 中，函数可分为严格确定、确定和非确定 3 类。

- 如果对于一组特定的输入值，函数始终返回相同的结果，则该函数就是严格确定的。
- 而对于用户定义的函数，与系统函数不同，判断其是否确定的标准相对宽松。如果对于一组特定的输入值和数据库状态，用户定义函数始终返回相同的结果，则

该用户定义的函数就是确定的。例如定义一个用于数据访问的函数，对于数据库表的内容不同，其返回的结果会不同，所以它不是严格确定的，当时对于相同的数据库状态，使用相同的输入值始终返回相同的结果，则也可以从这个角度认为它是确定的。

- 使用同一组输入值重复调用非确定性函数，返回的结果可能会不同。例如，函数 GETDATE()返回当前的日期时间，在不同的时间调用返回不同的结果，所以是非确定的。SQL Server 对各种类型的非确定性函数进行了限制。因此，应慎用非确定性函数。

对于 SQL Server 内置的系统函数，确定性和严格确定性是相同的。而对于使用 T-SQL 定义的用户定义函数，系统将验证定义并防止定义非确定性函数。进行数据访问或未绑定到架构的用户定义函数被视为非严格确定性函数。对于公共语言运行时（CLR）函数，函数定义将指定该函数的确定性、数据访问和系统数据访问等属性。但是由于这些属性未经系统验证，因而函数将始终被视为非严格确定性函数。

只有确定性函数才可以在索引视图、索引计算列、持久化计算列或 T-SQL 用户定义函数的定义中调用，而缺少确定性的函数将禁止在这些情况下使用。

2.8.2　聚合函数

聚合函数对一组值执行计算，并返回单个值。除了 COUNT 以外，聚合函数都会忽略空值。聚合函数经常与 SELECT 语句的 GROUP BY 子句一起使用。所有聚合函数均为确定性函数。这表示任何时候使用一组特定的输入值调用聚合函数，所返回的值都是相同的。

聚合函数可以在 SELECT 语句的选择列表（子查询或外部查询）、COMPUTE 或 COMPUTE BY 子句，以及 HAVING 子句中使用。

T-SQL 提供的最常用的聚合函数有以下几种。

- AVG()：返回组中各值的平均值。
- MIN()：返回表达式中的最小值。
- MAX()：返回表达式中的最大值。
- SUM()：返回表达式中所有值的和或仅非重复值的和。SUM()只能用于数字列。空值将被忽略。
- COUNT()和 COUNT_BIG()：返回组中的项数。COUNT()与 COUNT_BIG()函数类似。两个函数唯一的差别是它们的返回值。COUNT()始终返回 int 数据类型值，COUNT_BIG()始终返回 bigint 数据类型值。
- STDEV()：返回指定表达式中所有值的标准偏差。
- STDEVP()：返回指定表达式中所有值的总体标准偏差。
- VAR()：返回指定表达式中所有值的方差。
- VARP()：返回指定表达式中所有值的总体方差。

可能对于一般的业务来说，开发人员使用的最多的就是 COUNT()、SUM()、MIN()、MAX()和 AVG()函数了。比如要查看 AdventureWorks 数据库中现有产品种类数以及产品的

标准价的平均值、最大值和最小值，则可以使用聚合函数进行查询，如代码 2.48 所示。

代码 2.48　使用聚合函数

```
USE AdventureWorks2012
GO
SELECT COUNT(*) AS Pcount,AVG(StandardCost) Pavg,MIN(StandardCost) Pmin,
MAX(StandardCost) Pmax
FROM Production.Product
```

说明：COUNT()函数必须带一个参数，但是这个参数不一定是列名或者*，也可以是完全没有意义的数字、字符串、日期等类型的数据，使用这些参数也并不影响COUNT()函数返回的结果。

2.8.3　日期和时间函数

日期函数用于显示关于日期和时间的信息。使用这些函数可更改 datetime 和 smalldatetime 值，还可以对它们执行算术运算。可将日期函数用于可以使用表达式的任何地方。

日期函数中常用的函数有以下几种。

- DATEADD()：返回给指定日期加上一个时间间隔后的新 datetime 值。
- DATEDIFF()：返回跨两个指定日期的日期边界数和时间边界数。
- DATENAME()：返回表示指定日期的指定日期部分的字符串。
- DATEPART()：返回表示指定日期的指定日期部分的整数。除了用做 DATEPART (dw，date)外都具有确定性。dw 是 weekday 的日期部分，取决于设置每周的第一天的 SET DATEFIRST 所设置的值。
- DAY()：返回一个整数，表示指定日期的天 datepart 部分。
- GETDATE()：以 datetime 值的 SQL Server 标准内部格式返回当前系统日期和时间。
- GETUTCDATE()：返回表示当前的 UTC 时间（通用协调时间或格林尼治标准时间）的 datetime 值。当前的 UTC 时间来自当前的本地时间和运行 Microsoft SQL Server 实例的计算机操作系统中的时区设置。
- MONTH()：返回表示指定日期的“月”部分的整数。
- YEAR()：返回表示指定日期的年份的整数。

日期时间函数在实际开发中使用的最多的可能就是 GETDATE()函数了，该函数返回当前的日期时间，可以用做数据表中 CreateTime 字段的默认值。用户只要向数据表中插入数据，该字段就可以使用 GETDATE 记录下当前的时间，也可以用于计算人的年龄等。例如要查看当前所有员工的年龄，并根据员工的年龄从小到大进行排序，其 SQL 脚本如代码 2.49 所示。

代码 2.49　使用日期函数

```
USE AdventureWorks2012;
```

```
GO
SELECT e.EmployeeID, YEAR(GETDATE())-YEAR(e.BirthDate) AS Age
FROM HumanResources.Employee e
ORDER BY Age
```

另外，还有一个十分有用的函数 DATEPART()，输入参数 datepart 是指定要返回的日期部分的参数。DAY()、MONTH()和 YEAR()函数分别是 DATEPART(dd，date)、DATEPART(mm，date)和 DATEPART(yyyy，date)的同义词。表 2.8 列出了 Microsoft SQL Server 可识别的日期部分及其缩写。

表 2.8　DATEPART函数的参数

日期部分	缩　　写	说　　明	日期部分	缩　　写	说　　明
year	yy, yyyy	年	weekday	dw	第几个星期
quarter	qq, q	季度	hour	hh	小时
month	mm, m	月	minute	mi, n	分
dayofyear	dy, y	一年中的第几天	second	ss, s	秒
day	dd, d	日期	millisecond	ms	毫秒
week	wk, ww	星期			

例如要查看当前的年份和当前时间是本年内的第几周，则可以使用 DATEPART()函数。

```
SELECT DATEPART(yy,GETDATE()),DATEPART(wk,GETDATE())
```

注意：如果只指定年份的后两位数字，则小于或等于 two-digit year cutoff 配置选项值的后两位数字的值将与截止年份处于同一世纪中。比此选项值的后两位数字大的值先于截止年份的世纪。例如，如果 two-digit year cutoff 是 2049（默认值），则 49 将被解释为 2049，而 50 则将被解释为 1950。为了避免产生歧义，请使用四位数字表示年份。

2.8.4　数学函数

数学函数对数字表达式进行数学运算并返回运算结果。数学函数对 SQL Server 2008 系统提供的数字数据进行运算，这些数据类型包括 decimal、integer、float、real、money、smallmoney、smallint 和 tinyint。默认情况下，对 float 数据类型数据的内置运算的精度为 6 个小数位数。

表 2.9 列出了标量函数，这些函数通常基于作为参数提供的输入值执行计算，并返回一个数值。这些函数的功能与 C 语言、C#语言等编程语言中的函数相同，这里就不一一解释了，若读者不知道某函数的功能和用法可以查看 SQL Server 帮助文档。

表 2.9　数学函数

函　　数	说　　明
ABS()	求绝对值

续表

函　　数	说　　明
ACOS()	反余弦
ASIN()	反正弦
ATAN()	反正切
ATN2()	返回以弧度表示的角
CEILING()	返回大于或等于指定数值表达式的最小整数
COS()	余弦
COT()	余切
DEGREES()	返回以弧度指定的角的相应角度
EXP()	返回数值表达式以 e 为底的指数
FLOOR()	返回小于或等于指定数值表达式的最大整数
LOG()	返回数值表达式以 10 为底的对数
LOG10()	返回指定表达式的常用对数
PI()	圆周率
POWER()	幂函数
RADIANS()	对于在数值表达式中输入的度数值返回弧度值
RAND()	返回从 0～1 之间的随机数
ROUND()	返回一个数值，舍入到指定的长度或精度
SIGN()	返回数值表达式的符号：正号（+1）、负号（–1）或零（0）
SIN()	正弦
SQRT()	开方函数
SQUARE()	返回数值表达式的平方
TAN()	正切

说明： 算术函数（例如 ABS()、CEILING()、DEGREES()、FLOOR()、POWER()、RADIANS()和 SIGN()）将根据输入值的类型返回与输入值具有相同数据类型的值。三角函数和其他函数（包括 EXP()、LOG()、LOG10()、SQUARE()和 SQRT()）将输入值转换为 float 并返回 float 值。

除 RAND()以外的所有数学函数都为确定性函数。这意味着在每次使用特定的输入值集调用这些函数时，它们都将返回相同的结果，仅当指定种子参数时 RAND()才是确定性函数。

一般业务系统中像 SIN()、SQRT()等三角函数及开方函数并不常用，在数学函数中使用较多的是取绝对值函数 ABS()、随机函数 RAND()和 3 个舍入函数 ROUND()、FLOOR()、CEILING()。

ABS()是返回指定数值表达式的绝对值（正值）的数学函数。如运行下列语句：

```
SELECT ABS(-1.0), ABS(0.0), ABS(1.0)
```

将返回这 3 个数的绝对值 1.0、0.0 和 1.0。

RAND()返回从 0～1 之间的随机 float 值。RAND()函数可以接受一个参数作为产生随

机数的种子。对于相同的种子将产生相同的随机数。

对于一个连接，如果使用指定的种子值调用 RAND()，则 RAND()的所有后续调用将基于使用该指定种子值的 RAND()调用生成结果。例如运行：

```
SELECT RAND(100),RAND(),RAND()
```

虽然后面两个 RAND()函数没有指定种子，但是无论执行多少次，这 3 个 RAND()函数产生的随机数都不变。

ROUND()函数的作用就是四舍五入，不过 ROUND()函数的第二个参数用于指定舍入的进度，即保留几位小数的精度。如果为负数，则表示向整数方向舍入。例如运行：

```
SELECT ROUND(745.4,-1),ROUND(745.4,-2)
```

将得到结果 750.0，700.0。

注意：ROUND()虽然对数字进行了四舍五入，但是小数位数仍然不变。所以 ROUND(745.4,-1)得到的是 750.0 而不是 700。

FLOOR()返回小于或等于指定数值表达式的最大整数，即向下取整。如 FLOOR(123.4)和 FLOOR(123.7)都将返回 123。

CEILING()正好与 FLOOR()相反，返回大于或等于指定数值表达式的最小整数，即向上取整。如 CEILING(123.4)和 CEILING(123.7)都返回 124。

2.8.5　字符串函数

字符串函数用于对字符和二进制字符串进行各种操作。它们返回对字符数据进行操作时通常所需要的值。这里的字符串包括字符串类型和 Unicode 字符串类型，大部分字符串函数只能用于这两种 SQL 数据类型，或用于可被隐式转换为字符串数据类型的数据类型。只有某些字符串函数还可用于 binary 和 varbinary 数据类型。使用字符串函数可以执行下列操作：

- 仅检索字符串的一部分（SUBSTRING()函数）。
- 搜索字符串的发音相似性（SOUNDEX()函数和 DIFFERENCE()函数）。
- 查找列或表达式中特定字符串的开始位置。例如，要查找字母 W 在“Hello World”中的位置号。
- 将多个字符串连接成一个字符串。例如，将姓字符串和名字符串连接成一个姓名字符串。
- 将非字符串值转换为字符串值（例如，将以 float 类型存储的值 123.45 转换为 char 类型）。
- 将特定字符串插入现有字符串。例如，将字符串“BC”插入现有字符串“ADEFG”中，以生成字符串“ABCDEFG”。

表 2.10 列出了所有的内置字符串函数。所有内置字符串函数都是具有确定性的函数。这意味着每次用一组特定的输入值调用它们时，都返回相同的值。

表 2.10　字符串函数

函　　数	说　　明
ASCII()	返回字符表达式中最左侧的字符的 ASCII 代码值
CHAR()	将 int ASCII 代码转换为字符
CHARINDEX()	返回字符串在另一个字符串中的起始位置
DIFFERENCE()	返回一个整数值，指示两个字符表达式的 SOUNDEX 值之间的差异
LEFT()	返回字符串中从左边开始指定个数的字符
LEN()	返回字符串长度
LOWER()	返回字符串的小写形式
LTRIM()	返回删除了前导空格的字符表达式
NCHAR()	返回具有指定的整数代码的 Unicode 字符
PATINDEX()	返回指定表达式中某模式第一次出现的起始位置
QUOTENAME()	返回带有分隔符的 Unicode 字符串
REPLACE()	将表达式中的一个字符串替换为另一个字符串或空字符串
REPLICATE()	返回多次复制后的字符表达式
REVERSE()	按相反顺序返回字符表达式
RIGHT()	返回字符表达式中从右端开始到指定字符位置的部分
RTRIM()	返回删除了尾随空格的字符表达式
SOUNDEX()	返回一个由四个字符组成的代码，用于评估两个字符串的相似性
SPACE()	返回由重复的空格组成的字符串
STR()	返回由数字数据转换来的字符数据
STUFF()	将字符串插入另一字符串
SUBSTRING()	返回字符、二进制字符串或文本字符串的一部分
UNICODE()	返回输入表达式的第一个字符的整数值
UPPER()	返回字符串的大写形式

这些字符串函数在实际应用中都会被经常用到，尤其是其中的 LEFT()、RIGHT()、REPLACE()、CHARINDEX()、LEN()和 SUBSTRING()。这里简要介绍一下这几个函数，如果需要了解其他函数，可以参考 SQL Server 的帮助文档。

- LEFT()函数返回字符串中从左边开始指定个数的字符。如 LEFT('abcdefg',3)将返回前 3 个字符'abc'。
- RIGHT()函数返回字符串中从右边开始指定个数的字符。如 RIGHT('abcdefg',3)将返回后 3 个字符'efg'。
- REPLACE()函数用第 3 个表达式替换第一个字符串表达式中出现的所有第 2 个指定字符串表达式的匹配项。如 REPLACE('abcdefg','cd','xx')将返回'abxxefg'。
- LEN()函数返回指定字符串表达式的字符（而不是 B）数，其中不包含尾随空格。如 LEN('abcdefg')和 LEN('abcdefg   ')都将返回整数 7。
- CHARINDEX()函数返回字符串中指定表达式的开始位置。其语法为：

```
CHARINDEX ( expression1 ,expression2 [ , start_location ] )
```

其中 expression1 是一个表达式，其包含要查找的字符的序列。expression2 也是一个表

达式，通常是一个为指定序列搜索的列。expression2 属于字符串数据类别。start_location 是开始在 expression2 中搜索 expression1 时的字符位置。如果 start_location 未被指定、是一个负数或 0，则将从 expression2 的开头开始搜索。start_location 可以是 bigint 类型。

如果在 expression2 内找不到 expression1，则 CHARINDEX 返回 0。例如，在'abcdabcd'中找到第一个 b 和第二个 b 的位置的 SQL 为：

```
SELECT CHARINDEX('b','abcdabcd') --找第 1 个 b
SELECT CHARINDEX('b','abcdabcd',CHARINDEX('b','abcdabcd')+1)  --找第 2 个 b
```

返回的结果是 2，6。

注意：CHARINDEX()函数将第一个字符的位置作为 1，而不是像 C 语言那样将第一个字符的位置作为 0。

- SUBSTRING()函数用于返回字符表达式、二进制表达式、文本表达式或图像表达式的一部分。例如，SUBSTRING('abcdefg',3,2)将返回第 3 个字符后的两个字符 cd。

该函数一般与 CHARINDEX()函数和 LEN()函数一起使用，实现截取部分字符串。例如在员工表中，员工的登录名是采用“域名\用户名”的形式保存。若希望从中提取出用户名，则需要使用 SUBSTRING()函数。具体方法如代码 2.50 所示。

代码 2.50　使用 SUBSTRING()函数

```
USE AdventureWorks2012;
GO
SELECT e.LoginID,
SUBSTRING(e.LoginID,CHARINDEX('\',e.LoginID)+1,LEN(e.LoginID)-CHARINDEX
('\',e.LoginID)) AS UserLoginName
FROM HumanResources.Employee e
```

2.8.6　其他常用函数

除了以上几种常用的函数类型外还有以下函数也经常使用，这里也一并列举出来。这些函数在接下来的章节中也会频繁出现。

NEWID()函数返回一个 GUID（全球唯一标识符），主要用于将 GUID 作为主键的字段中，将 GUID 字段的默认值设置为 NEWID()即可。

OBJECT_ID()函数返回架构范围内对象的数据库对象标识号。如果对象不存在，则返回 NULL。该函数常用于判断某个数据库对象是否存在，并根据判断结果决定执行的 SQL 语句。例如，在创建一个表的时候必须先判断数据库中是否已经存在该表。如果存在，则必须先删除该表后重新创建表；否则系统将抛出异常。使用 OBJECT_ID()函数协助创建 Student 表的 SQL 脚本，如代码 2.51 所示。

代码 2.51　使用 OBJECT_ID 协助创建表

```
IF(OBJECT_ID('dbo.Student') IS NOT NULL)    --判断 Student 表已经存在
    DROP TABLE dbo.Student                  --删除表
GO
```

```
CREATE TABLE Student --创建表
(
    Sid int IDENTITY PRIMARY KEY,
    Sname nvarchar(10) NOT NULL
)
```

CAST()和 CONVERT()函数用于将一种数据类型的表达式显式转换为另一种数据类型的表达式。关于 SQL Server 中众多数据类型之间的转换笔者就不在此讲解了，在 SQL Server 帮助文档中给出了详细的介绍。这里主要介绍一下最常用的转换：字符串转换为数字和日期转换为字符串。如需要将字符串'123'转换为数字 123 对应的脚本为：

```
SELECT CONVERT(int ,'123')
SELECT CAST('123' AS int)
```

这里 CONVERT()和 CAST()的作用是一样的，只是语法不同而已。如果要将当期的时间 DateTime 类型转换为字符串，以 yyyyMMdd 的形式表示则脚本为：

```
SELECT CONVERT(char(8),GETDATE(),112)
```

这里 112 表示时间转换为字符串的样式为 yyyyMMdd 的形式。关于其他的样式用户可以查询帮助手册。

ISNULL()函数使用指定的替换值替换 NULL。ISNULL()函数的语法格式为：

```
ISNULL ( check_expression , replacement_value )
```

其中，check_expression 是将被检查是否为 NULL 的表达式。check_expression 可以为任何类型。当 check_expression 为 NULL 时该函数将返回 replacement_value 的表达式。

注意：replacement_value 的数据类型必须是可以隐式转换为 check_expresssion 的类型，否则系统将会报错。

代码 2.52 查找所有产品的重量平均值，用值 50 替换 Product 表的 Weight 列中的所有 NULL 项。

代码 2.52 使用 ISNULL()函数

```
USE AdventureWorks2012;
GO
SELECT AVG(ISNULL(Weight, 50)) --使用 50 代替 NULL 值
FROM Production.Product;
```

2.9 小 结

本章主要讲解了 T-SQL 查询语言的基础知识。T-SQL 按照功能的不同，可以分为数据定义语言、数据操纵语言、数据查询语言和数据控制语言 4 种。SQL 对数据库的最基本的操作是 CRUD，而在数据库操作中查询是最为复杂，也是最为常用的操作。

在多个表之间的联接查询有内联接、外联接、完全联接和交叉联接。另外，还介绍了 SQL Server 中支持的各种数据类型，以及在 SQL 开发中使用变量、操作符、流程控制和函数等实现 T-SQL 脚本编程，将数据处理和逻辑判断通过 T-SQL 来实现等内容。

第 3 章　数据库基本操作

在成功连接数据库系统后，将面临数据库的创建、表的创建、数据的录入、修改和查询、视图的创建和查询，以及存储过程的操作等基本操作。本章主要讲解 T-SQL 的基本操作，以及使用 T-SQL 语句和 SSMS 进行数据库基本操作。T-SQL 的高级特性将在开发篇进行深入讲解。

3.1　数据库操作

数据库是表、视图、存储过程和函数等数据库对象的容器。本节主要介绍数据库的一些概念和如何创建、修改和删除数据库。

3.1.1　创建数据库

SQL Server 中提供了 CREATE DATABASE 语句来创建数据库，其语法格式为：

```
CREATE DATABASE database name
[additional parameters]
```

若要创建一个数据库 TestDB，最简单的写法为：

```
CREATE DATABASE TestDB
```

运行该命令后，SQL Server 会在默认的数据库文件目录（C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data）下创建数据库文件 TestDB.mdf 和日志文件 TestDB_log.ldf。数据库文件的大小、数据库中的数据库对象，以及数据库的其他属性都是从 model 数据库继承而来的。

注意：在 SSMS 中使用 T-SQL 创建的数据库，以及其他对象都不会立即展示在对象资源管理器中，用户只有刷新对象资源管理器才能看到。

当然，使用 T-SQL 语句也可以指定创建数据库的文件路径，以及文件的大小、增长量和最大文件限制等属性。如代码 3.1 所示，创建了一个数据库 Sales，该数据库数据文件在 C:\saledat.mdf 下，初始大小为 10MB，自动增长为 5MB，数据文件最大限制为 50MB。日志文件存放在 C:\salelog.ldf 下，初始大小 5MB，自动增长为 5MB，日志文件最大限制为 25MB。

代码 3.1　创建数据库

```
CREATE DATABASE Sales                  --创建 Sales 数据库
ON
( NAME = Sales dat,
    FILENAME = 'C:\saledat.mdf',       --数据文件路径
    SIZE = 10,                         --数据文件初始大小
    MAXSIZE = 50,                      --数据文件最大大小
    FILEGROWTH = 5 )                   --文件增长
LOG ON
( NAME = Sales_log,
    FILENAME = 'C:\salelog.ldf',       --日志文件路径
    SIZE = 5MB,                        --日志文件初始大小
    MAXSIZE = 25MB,                    --日志文件最大大小
    FILEGROWTH = 5MB )                 --文件增长
```

除了使用 T-SQL 语句创建数据库外，SSMS 也提供了可视化的界面来创建数据库，其主要操作如下所述。

（1）在对象资源管理器中右击“数据库”，在弹出的快捷菜单中选择“新建数据库”选项，系统将弹出“新建数据库”对话框，如图 3.1 所示。

（2）在“数据库名称”文本框中输入需要创建的数据库名，系统将自动设置数据库文件的逻辑名称。

（3）若有必要，用户可以设置数据库文件的初始大小、自动增长和路径等属性。在设置自动增长属性时将弹出“更改 TestDB1 的自动增长设置”对话框，如图 3.2 所示。

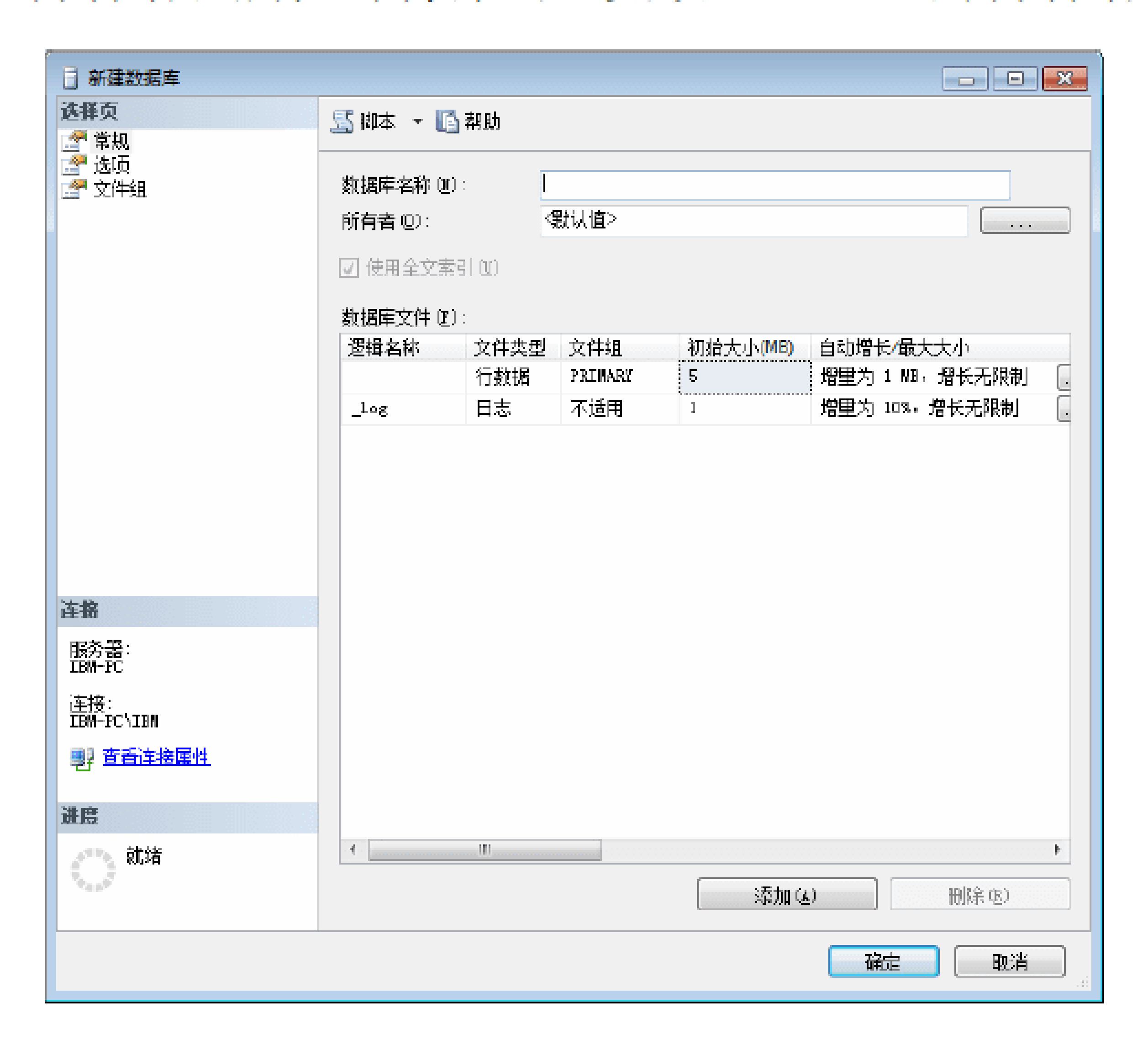


图 3.1　“新建数据库”对话框

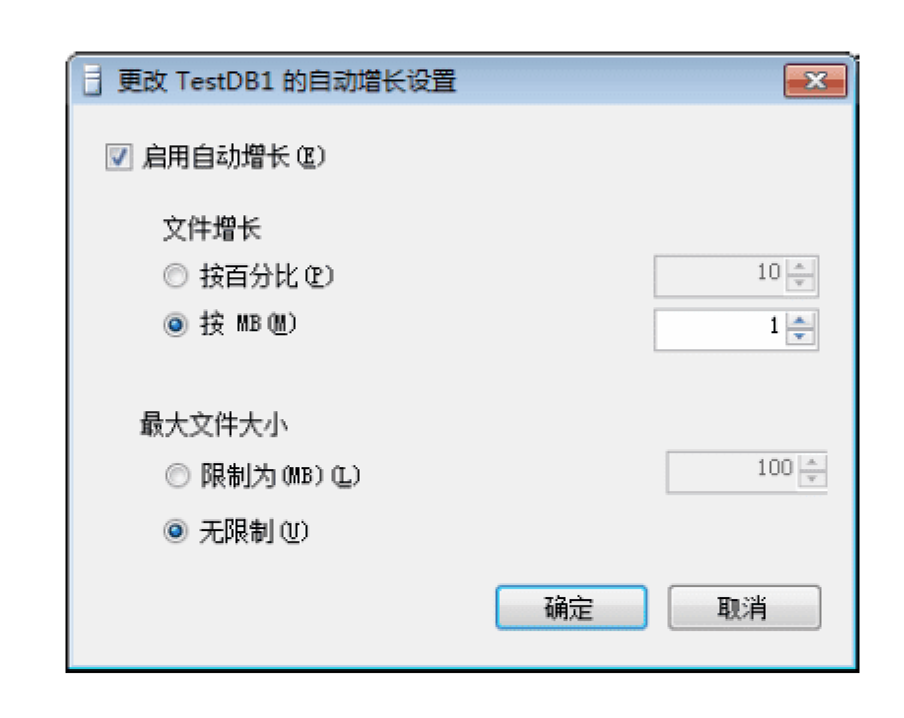


图 3.2　“更改 TestDB1 的自动增长设置”对话框

随着数据量的增加，初始的数据文件大小将无法容纳更多的数据。在启用自动增长后，当现有的文件大小小于数据需要的存储空间大小时，系统将按照百分比或固定增长量加大文件的数据空间。另外，用户还可以设置文件的最大值。当文件不断增长到该最大值后将无法再增大，当然也无法保存更多的数据。

（4）在设置好这些属性后单击“确定”按钮，系统将创建该数据库，并同时刷新对象资源管理器。

🔔注意：一般情况下不要限制文件增长的大小。当文件到达限制的大小时将无法保存更多的数据时，DBMS 将直接抛出异常。

3.1.2　修改数据库

SQL Server 提供了 ALTER DATABASE 命令用于修改现有数据库，其格式为：

```
ALTER DATABASE database name
[additional parameters]
```

通过使用 ALTER DATABASE 命令，可以更改数据库的文件路径、初始大小、自动增长以及各种数据库选项。关于数据库的选项将会在后面的章节进行讲解，本节只涉及数据库的基础修改。若用户希望对各个数据库选项有所了解可以查看 SQL Server 的联机丛书。

在 3.1.1 节中创建好数据库 Sales 后，如果需要将该数据库的数据文件初始大小修改为 20MB，这时可以使用代码 3.2 对该数据库进行修改。

代码 3.2　修改数据库初始大小

```
ALTER DATABASE Sales
MODIFY FILE
    (NAME = Sales_dat,
   SIZE = 20MB); --修改初始大小为 20M
```

除了使用 T-SQL 脚本对数据库更改外，SSMS 也支持对数据库的更改。其操作方式为：

（1）在 SSMS 的对象资源管理器中，右击需要修改的数据库。在弹出的快捷菜单中选择“属性”选项，系统将弹出“数据库属性”对话框，如图 3.3 所示。

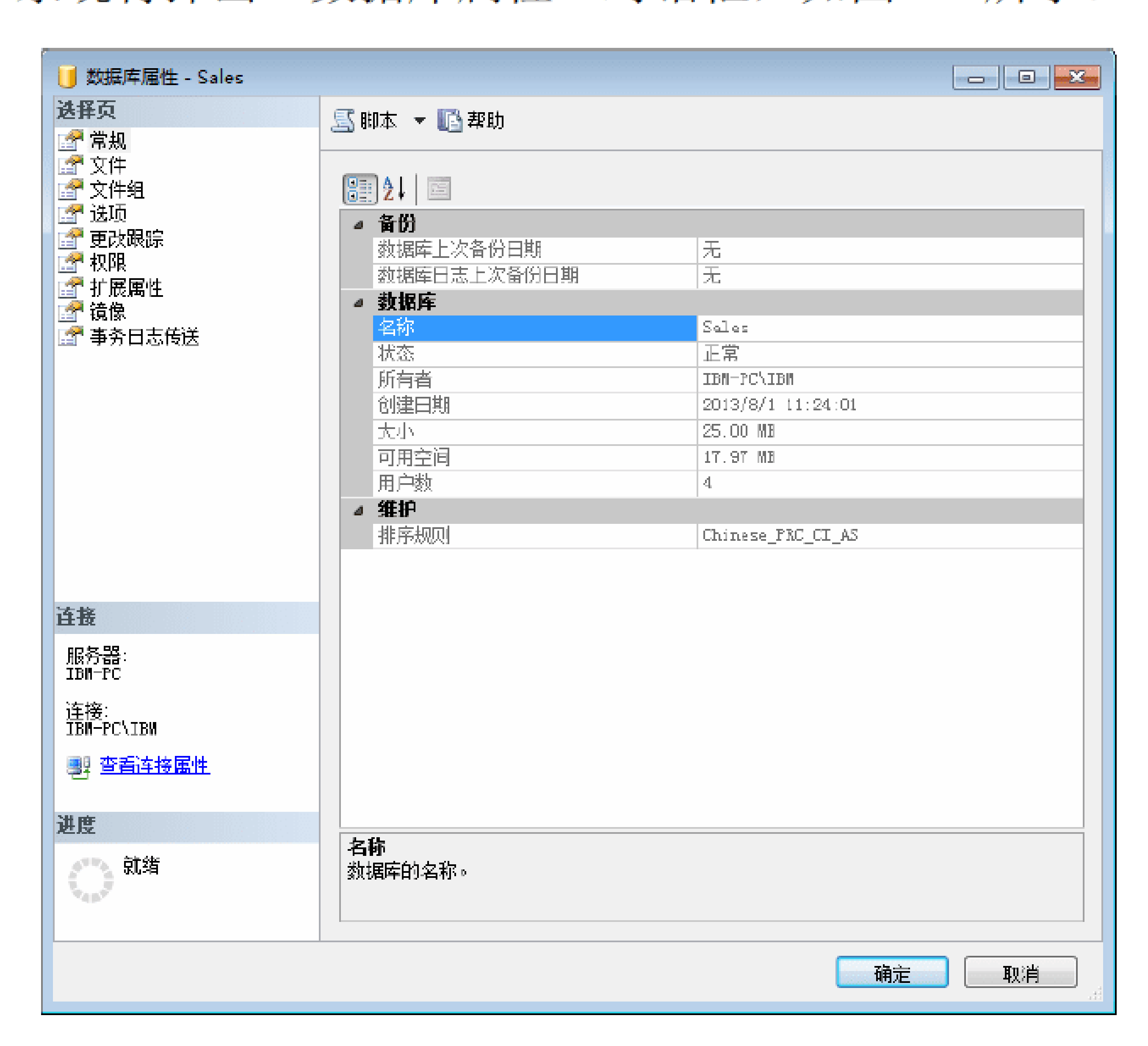


图 3.3　“数据库属性”对话框

（2）选择选项页中的“文件”选项，右侧窗口将显示数据库的文件配置，如图 3.4 所示。

（3）图 3.4 显示的是通过代码创建的数据库 Sales 的属性。用户可以在该窗口修改该数据库的初始大小和自动增长属性。

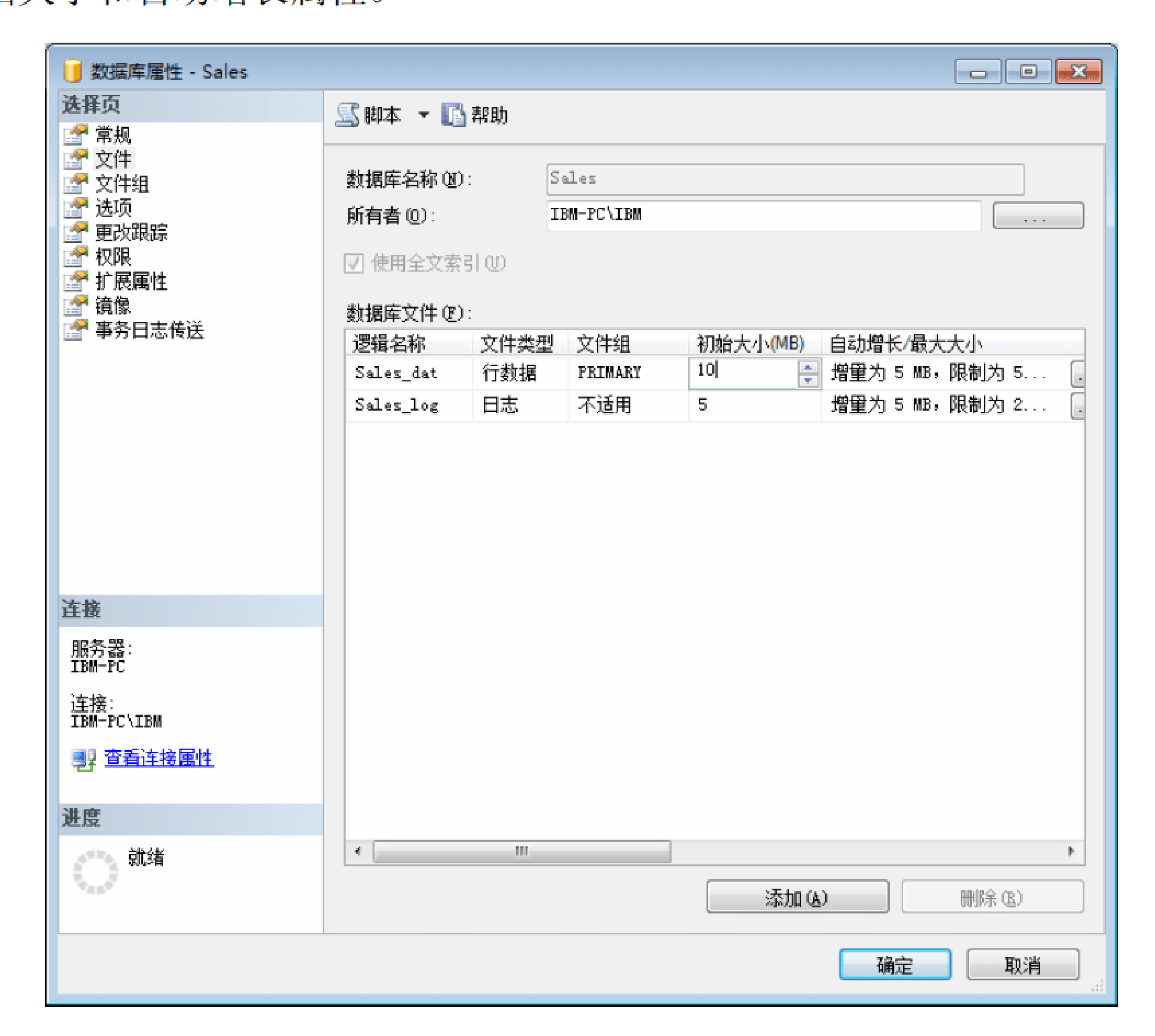


图 3.4　数据库的文件配置

（4）修改完成后单击“确定”按钮，系统将把更改应用到数据库中。

SSMS 的一大特点就是可以将界面上的更改生成 T-SQL 脚本提供给用户。在修改了初始大小和自动增长后，单击“脚本”按钮旁的下拉按钮将显示下拉列表框，如图 3.5 所示。

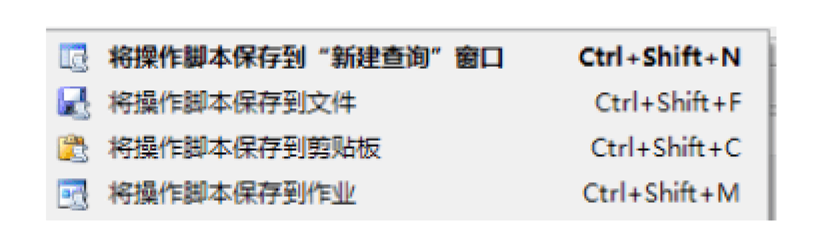


图 3.5　脚本下拉列表框

在下拉列表中选择“将操作脚本保存到新建查询窗口”选项，或者使用提示的快捷键 Ctrl+Shift+N 便可生成操作脚本，如代码 3.3 所示。

代码 3.3　SSMS 生成的 SQL 脚本

```
USE [master]
GO
ALTER DATABASE [Sales] MODIFY FILE ( NAME = N'Sales_dat', SIZE = 10240KB )
GO
```

用户可以在属性窗口中单击“确定”按钮或将生成的脚本在 SSMS 中运行。但是不要既单击“确定”按钮又运行生成的脚本，这将造成重复操作。

3.1.3　删除数据库

当创建好的数据库由于某种原因不再使用而需要将数据库删除时，SQL Server 提供了 DROP DATABASE 命令。其使用格式为：

```
DROP DATABASE { database_name | database_snapshot_name } [ ,...n ]
```

在删除数据库之前必须保证被删除的数据库当前未被使用。删除数据库操作将会把数据库文件（包括数据文件和日志文件）从磁盘物理删除。删除后使用一般手段将无法恢复被删除的数据库，所以在进行删除操作之前一定要一再确认。若要将前面创建的 Sales 数据库删除，则删除数据库的脚本为：

```
USE master; --保证当前连接不是连接到 Sales 数据库
GO
DROP DATABASE Sales
```

另外，DROP DATABASE 支持一次删除多个数据库。读者可以先使用 CREATE DATABASE 命令创建数据库 Sales2，Sales3。现在需要一次删除这两个数据库，其代码为：

```
USE master;
GO
DROP DATABASE Sales2,Sales3
```

同样，SSMS 也提供了删除数据库的可视化操作。在对象资源管理器中右击需要删除的数据库。在弹出的快捷菜单中选择“删除”选项或者直接使用快捷键 Delete，系统将弹出删除对象窗口。在该窗口下方有两个复选框如图 3.6 所示。

☑ 删除数据库备份和还原历史记录信息(D)
☐ 关闭现有连接(C)

图 3.6　删除对象窗口的复选框

在删除时为了保证删除成功，一般要启用“关闭现有连接”复选框。然后单击“确定”按钮，系统将对数据库进行删除。

技巧： 在 SSMS 中删除数据库对象（包括数据库、表、列和约束等）时，都可以在对象资源管理器中选定该对象后使用 Delete 键删除选定对象。

3.2　表　操　作

在数据库中操作的最多的对象就是表。表是数据管理的基本单元，所有的业务数据都是以表为容器存放在数据库中。本节将主要讲解通过 T-SQL 语句和 SSMS 实现表的创建、修改和删除操作。

3.2.1　表简介

表是包含数据库中所有数据的数据库对象。表定义是一个列集合。数据库中的表与平

常使用的 Excel 相似，都是以行和列的形式进行组织的，数据存在于单元格中。一行数据就代表一条唯一的记录，而一个列则只记录一个字段。例如在一个学生管理系统中存在一个学生表，每一行数据就代表了一个学生，而表中的学号、姓名、生日等列则代表学生的属性信息。SQL Server 2012 中的表包括下列主要组件。

- ❑ 列：每一列代表由表建模的对象的某个属性。例如图 3.7 中的 LoginID 列、JobTitle 列和 Hiredate 列等。
- ❑ 行：每一行代表由表建模的对象的一个单独实例。例如图 3.7 中的每一行的数据就代表一个单独的员工。

BusinessEntit...	NationalIDN...	LoginID	OrganizationNo...	Organization...	JobTitle	BirthDate	MaritalStatus	Gender	HireDate	SalariedFlag
1	295847284	adventure-works\k...	/	0	Chief Executiv...	1963-03-02	S	M	2003-02-15	True
2	245797967	adventure-works\t...	/1/	1	Vice President...	1965-09-01	S	F	2002-03-03	True
3	509647174	adventure-works\r...	/1/1/	2	Engineering ...	1968-12-13	M	M	2001-12-12	True
4	112457891	adventure-works\r...	/1/1/1/	3	Senior Tool D...	1969-01-23	S	M	2002-01-05	False
5	695256908	adventure-works\g...	/1/1/2/	3	Design Engin...	1946-10-29	M	F	2002-02-06	True
6	998320692	adventure-works\j...	/1/1/3/	3	Design Engin...	1953-04-11	M	M	2002-02-24	True
7	134969118	adventure-works\d...	/1/1/4/	3	Research and...	1981-03-27	M	M	2003-03-12	True
8	811994146	adventure-works\d...	/1/1/4/1/	4	Research and...	1980-07-06	S	F	2003-01-30	True
9	658797903	adventure-works\g...	/1/1/4/2/	4	Research and...	1973-02-21	M	F	2003-02-17	True
10	879342154	adventure-works\...	/1/1/4/3/	4	Research and...	1979-01-01	M	M	2003-06-04	True
11	974026903	adventure-works\o...	/1/1/5/	3	Senior Tool D...	1972-02-18	S	M	2005-01-05	False
12	480168528	adventure-works\t...	/1/1/5/1/	4	Tool Designer	1953-08-29	M	M	2002-01-11	False
13	486228782	adventure-works\ja...	/1/1/5/2/	4	Tool Designer	1983-06-29	M	F	2005-01-23	False
14	42487730	adventure-works\...	/1/1/6/	3	Senior Design...	1973-07-17	S	M	2005-01-30	True
15	56920285	adventure-works\s...	/1/1/7/	3	Design Engin...	1955-06-03	M	F	2005-02-18	True
16	24756624	adventure-works\d...	/2/	1	Marketing M...	1969-04-19	S	M	2002-01-20	True
17	253022876	adventure-works\k...	/2/1/	2	Marketing As...	1981-06-03	S	M	2001-02-26	False
18	222969461	adventure-works\j...	/2/2/	2	Marketing Sp...	1972-04-06	S	M	2005-03-10	False
19	52541318	adventure-works\...	/2/3/	2	Marketing As...	1972-03-01	S	F	2005-03-17	False
20	323403273	adventure-works\w...	/2/4/	2	Marketing As...	1969-04-17	M	F	2005-02-07	False

图 3.7　Employee 表

在 SQL Server 中提供了 4 种类型的表可被用户操作。

- ❑ 持久基表：也就是平时使用的用来持久保存数据的表。持久基表是数据库中最常见的表，本文说的表或基表指的就是持久基表。
- ❑ 全局临时表：是在 tempdb 中创建的可被全局用户访问的临时表。全局临时表表名是以##开头的表，在创建后对任何用户和任何连接都是可见的。当引用该表的所有用户都与 SQL Server 实例断开连接后，将删除全局临时表。
- ❑ 局部临时表：是在 tempdb 中创建的只对创建者可见的临时表。局部临时表的表名以#开头。当创建者与 SQL Server 实例断开连接后，系统将自动删除局部临时表。
- ❑ 表变量：是在内存中创建的只对创建者可见的临时表，是 SQL Server 提供的一种数据类型。当创建者与 SQL Server 实例断开连接后，系统将自动删除表变量。

通过 SSMS 的对象资源管理器可以看到当前数据库系统中的持久基表、全局临时表和局部临时表。其中持久基表在每个数据库的表节点下，全局临时表和局部临时表都在 tempdb 数据库的“临时表”节点下。如图 3.8 所示。

3.2.2　使用 T-SQL 创建表

SQL Server 提供了 CREATE TABLE 命令用于表的创建。CREATE TABLE 的格式如代码 3.4 所示。

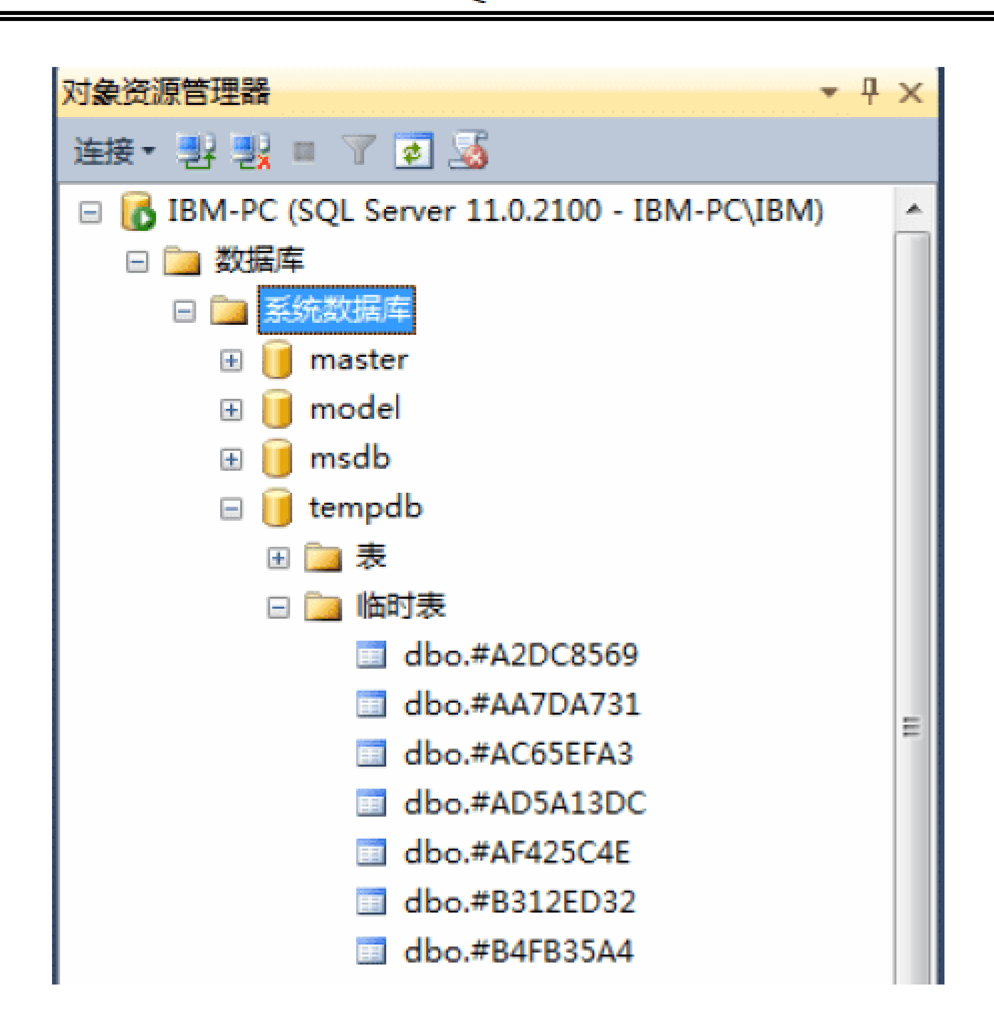


图 3.8　查看当前的“临时表”

代码 3.4　CREATE TABLE 的格式

```
CREATE TABLE
    [ database name . [ schema name ] . | schema name . ] table name
        ( { <column_definition> | <computed_column_definition> }
        [ <table constraint> ] [ ,...n ] )
    [ ON { partition scheme name ( partition column name ) | filegroup
        | "default" } ]
    [ { TEXTIMAGE_ON { filegroup | "default" } }
```

例如需要创建一个学生表 Student，该表中记录了学生的学号 Sid、姓名 Sname、生日 Sbirthday 和性别 Ssex，那么创建 Student 表的脚本如代码 3.5 所示。

代码 3.5　创建 Student 表

```
CREATE TABLE Student    --表名
(
    Sid char(10),        --各个字段名和字段类型
    Sname nvarchar(10),
    Sbirthday datetime,
    Ssex bit
)
```

代码 3.5 虽然可以创建学生表，但是这个表中既没有指定主键，也没有指定哪些字段能为空，哪些字段不能为空。这些都可以在创建表时进行设置，另外还需要对性别设置默认值。于是对代码 3.5 进行修改，修改后的脚本如代码 3.6 所示。

代码 3.6　修改后的 Student 表

```
CREATE TABLE Student
(
    Sid char(10) PRIMARY KEY,        --主键
    Sname nvarchar(10) NOT NULL,     --不为空
    Sbirthday datetime NULL,         --允许为空
    Ssex bit NOT NULL DEFAULT 0      --不为空，默认值是 0
)
```

在 SSMS 中运行该脚本后，刷新对象资源管理器便可以看到新建的表了。当然，在 CREATE TABLE 命令后不仅仅是跟主键、是否为空和默认值这几个属性，CREATE TABLE 命令还可以指定表的所在分区、外键、计算列、索引和约束等。这些高级设置将在使用到时再做讲解。

3.2.3　使用 SSMS 创建表

除了使用 CREATE TABLE 命令来创建表之外，SQL Server 2012 的 SSMS 也提供了可视化的界面来创建表。使用 SSMS 创建表的步骤如下所述。

（1）使用 SSMS 登录数据库并在对象资源管理器中展开需要创建表的数据库。

（2）右击“表”节点，在弹出的快捷菜单中选择“新建表”选项。系统将在主窗口中新建一个创建表的选项卡，如图 3.9 所示。

（3）同样以创建学生表为例，在第一个列名表格单元中输入 Sid，数据类型的表格单元中输入 char(10)，并取消“允许 Null 值”复选框的选中，如图 3.10 所示。这样学生表的学号字段就设置好了。

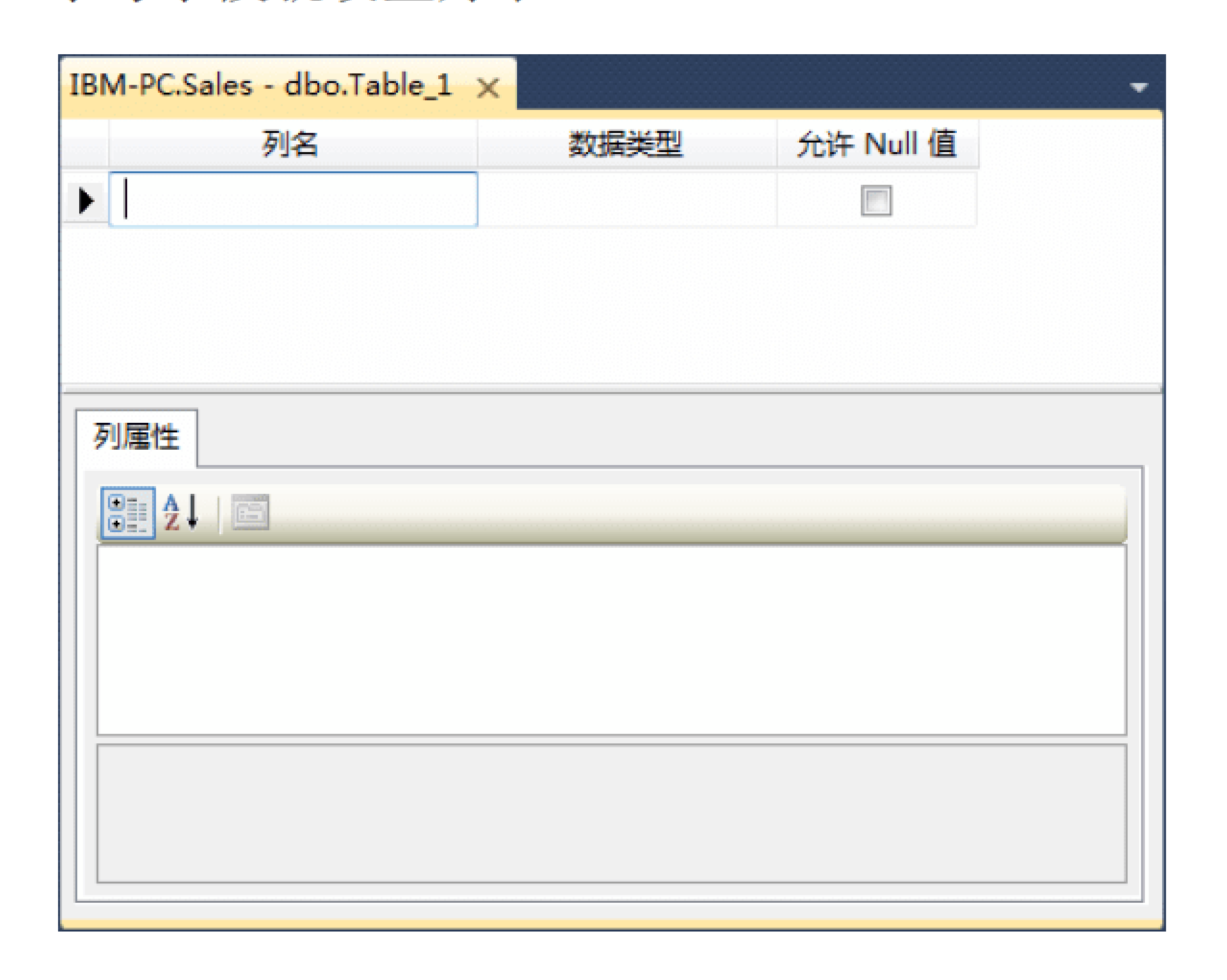



图 3.9　新建表的选项卡

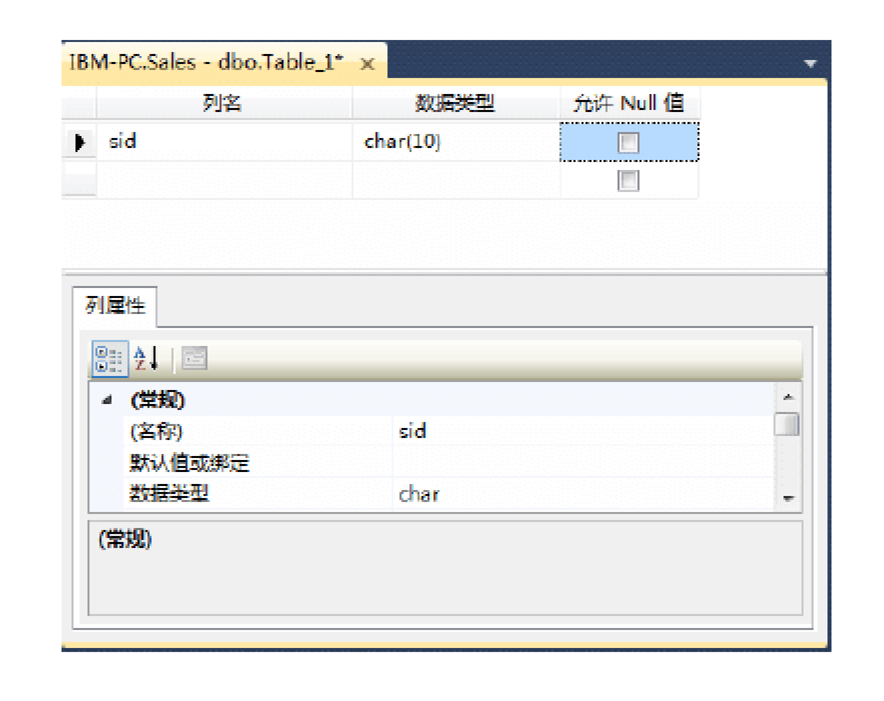


图 3.10　设置 Sid 字段

（4）使用同样的方法设置学生表的其他字段。对于 Ssex 设置默认值为 0，可以在建好 Ssex 列后将光标停留在 Ssex 列名上。下面的“列属性”选项卡中有“默认值或绑定”属性，在该属性中输入 0，默认值便设置完成了。设置完成后如图 3.11 所示。

（5）需要把 Sid 设置为表的主键。单击 Sid 左边的方格使 Sid 列选中。若需要选择多个列可以按 Ctrl 键再单击需要选择的列左边的复选框。在选中 Sid 的情况下，再选择工具栏中的“设置主键”按钮或者选择“表设计器”菜单下的“设置主键”命令，Sid 便被设置成了主键。设置成主键后，Sid 左边方格中有一个钥匙图案，如图 3.12 所示。

（6）使用 F4 键打开“属性”面板。在该面板中可以设置新建表的名称架构等。在“名称”属性中输入 Student，架构使用默认的 dbo 架构，如图 3.13 所示。

（7）在确保表名、架构和每一列都正确无误后，单击工具栏的“保存”按钮或者使用快捷键 Ctrl+S。此时表设计选项卡名后的“*”将消失，表示当前表已经保存。同时对象资

源管理器中“表”节点下便会出现新建的 Student 表，如图 3.14 所示，说明学生表创建成功。

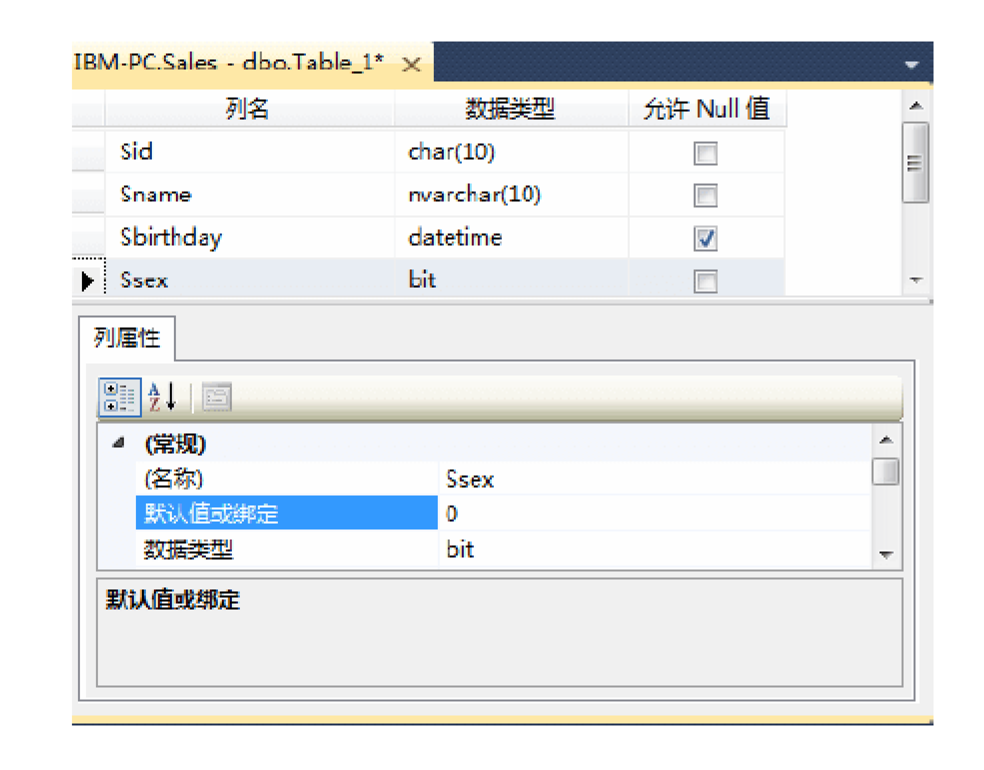


图 3.11　设置默认值

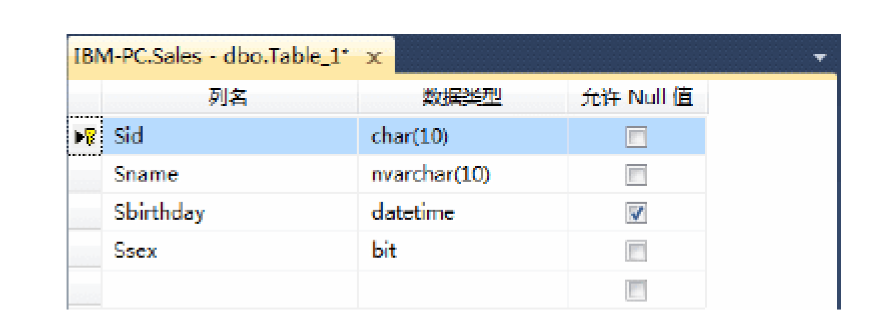


图 3.12　设置主键

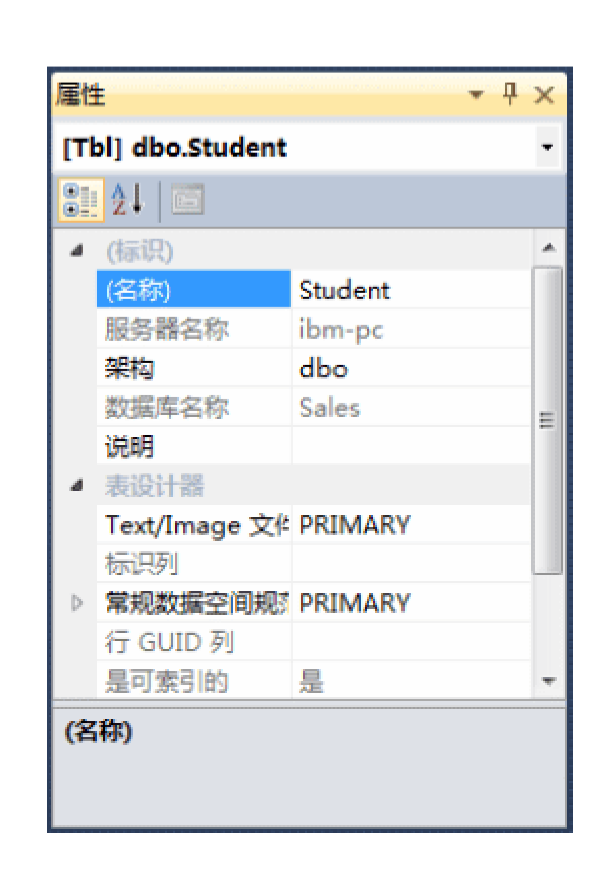


图 3.13　修改表的属性

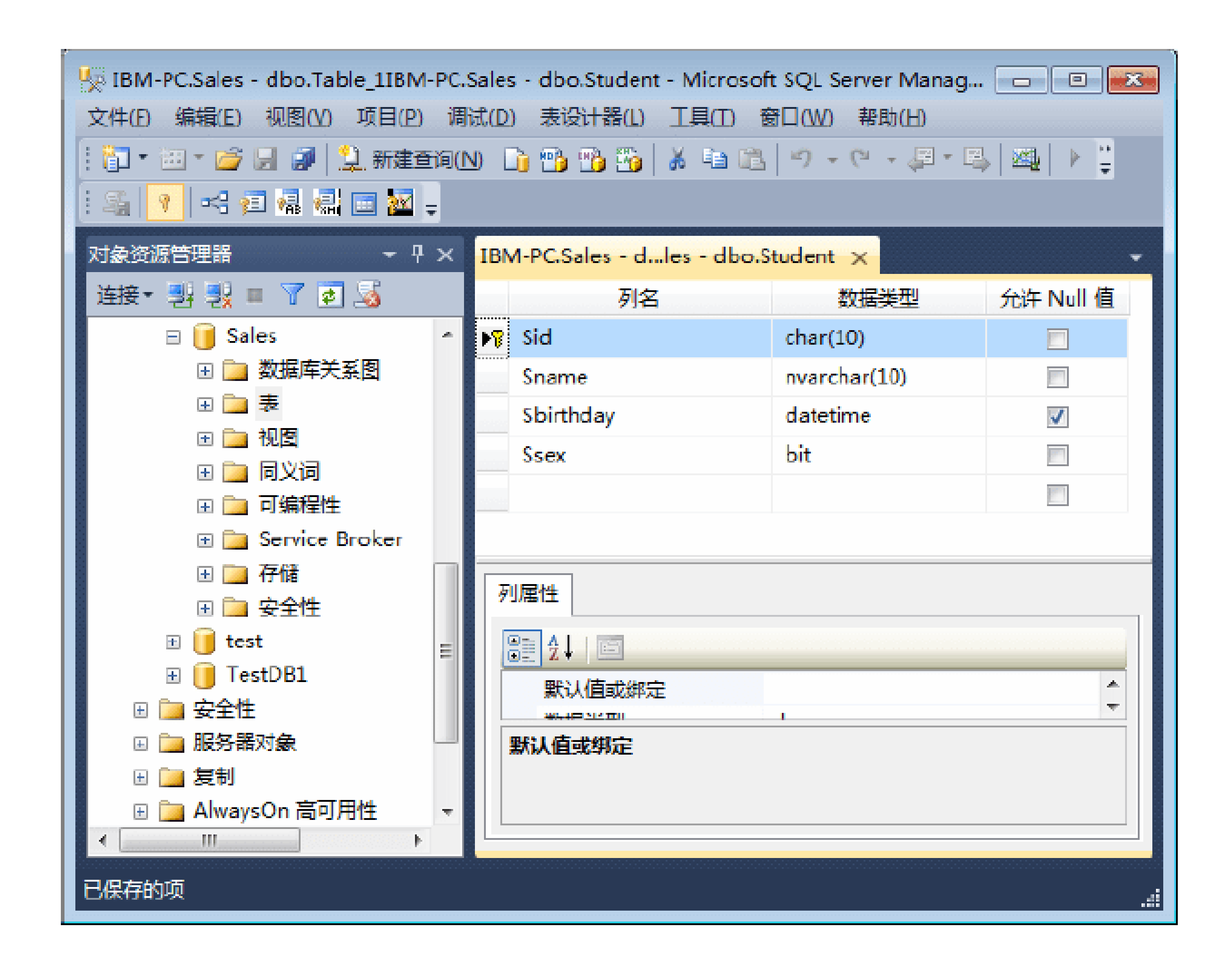


图 3.14　保存表

与 CREATE TABLE 命令一样，SSMS 的可视化界面也可以在创建表时进行更多高级属性的设置，在此暂不讲解。

3.2.4　创建临时表

SQL Server 中允许创建两种类型的临时表：全局临时表和局部临时表。当需要对同一数据集执行多个操作时，临时表是十分有用的。通过选择并将需要的数据结合到单个表中，可以避免让 DBMS 多次重复地提取并组合数据。临时表除了用于消除多次操作外，还可以提高执行速度。因为从一个表中读取数据比从多个表中读取数据要快。

在临时表的使用中一般以局部临时表居多。除了可以使用 CREATE TABLE 命令来创建表之外，T-SQL 还提供了一种通过数据转移的方式创建表的方法。SELECT…INTO…这种使用方式一般用来创建临时表，其使用方法如代码 3.7 所示。

代码 3.7　使用 SELECT INTO 创建表

```
USE AdventureWorks2012
GO
SELECT a.AddressID,a.City INTO #temp --使用 SELECT INTO 创建临时表
FROM Person.Address a
--从临时表中读取数据
SELECT *
FROM #temp
```

当然 SELECT INTO 命令并不是临时表的专利，在创建持久基表时也可以使用该命令。该命令既完成了表的创建工作，也完成了表的数据填充工作。

由于临时表的特性，一般情况下并不需要编写脚本对临时表执行删除操作。当连接断开时，系统将自动清除临时表。

注意： 使用 SELECT INTO 命令创建的表并不会拥有原表的各种约束信息，即 SELECT INTO 创建的表没有主键、默认值和 CHECK 约束等。若需要约束，则只有通过修改表来实现。

3.2.5　使用 T-SQL 修改表

由于当初的设计考虑不周或者是由于业务的变化等因素需要对表结构进行修改时，可以使用 T-SQL 中提供的 ALTER TABLE 命令。该命令可以用于添加、删除和更改表中的列，其格式如代码 3.8 所示。

代码 3.8　ALTER TABLE 的格式

```
ALTER TABLE [ database name . [ schema name ] . | schema name . ] table name
{
    ALTER COLUMN column name
    {
        [ type schema name. ] type name [ ( { precision [ , scale ]
            | max | xml schema collection } ) ]
        [ NULL | NOT NULL ]
        [ COLLATE collation name ]
    | {ADD | DROP } { ROWGUIDCOL | PERSISTED }
    }
    | [ WITH { CHECK | NOCHECK } ] ADD
    {
        <column_definition>
      | <computed_column_definition>
      | <table_constraint>
    } [ ,...n ]
    | DROP
    {
        [ CONSTRAINT ] constraint_name
        [ WITH ( <drop_clustered_constraint_option> [ ,...n ] ) ]
        | COLUMN column_name
    } [ ,...n ]
    | [ WITH { CHECK | NOCHECK } ] { CHECK | NOCHECK } CONSTRAINT
        { ALL | constraint name [ ,...n ] }
    | { ENABLE | DISABLE } TRIGGER
        { ALL | trigger_name [ ,...n ] }
```

```
    | SWITCH [ PARTITION source partition number expression ]
        TO [ schema name. ] target table
        [ PARTITION target partition number expression ]
}
```

1．使用ALTER TABLE添加列

向表中添加列是最常见的表修改操作。当使用 ALTER TABLE 语句向表中添加新列时，新列将出现在表的尾部。而查询时就会出现在查询结果的最右边。添加列的语法格式为：

```
ALTER TABLE table name
ADD column_name [DEFAULT <value>] [NOT NULL][IDENTITY][UNIQUE]
```

还是以前面提到的学生表为例。若希望在 Student 表中添加新的整型列 id，该列作为每行数据的标识，并且自动增加，那么就需要添加一个 IDENTITY 列。IDENTITY 列是 SQL Server 提供的一个自增标识列。每向表中添加一行数据时，该列的值将会自动增加。在数据完整性这一节中会对 IDENTITY 做详细介绍。

需要向 Student 表添加 IDENTITY 列 id，那么其脚本为：

```
ALTER TABLE Student
ADD id int IDENTITY
```

除非指定了默认值或 IDENTITY 标识，DBMS 对于已有行的新列都默认为 NULL。也就是说在添加行时，新行若是不能为空的，那么必须指定默认值。否则现有数据将会和 NOT NULL 约束冲突。

2．使用ALTER TABLE删除列

删除表中的列的语法格式为：

```
ALTER TABLE table name
DROP COLUMN column_name [,column_name2,…n]
```

同样以前面的学生表为例。先在该表中添加两个列 c1 和 c2，然后一次性将这两个列删除，其脚本如代码 3.9 所示。

代码 3.9　添加删除列

```
ALTER TABLE Student
ADD c1 int  --添加 c1
GO
ALTER TABLE Student
ADD c2 varchar(50)  --添加 c2
GO
ALTER TABLE Student
DROP COLUMN c1,c2  --删除 c1 和 c2
```

使用 ALTER TABLE 删除列时需要注意，如果一个列上有约束或者默认值，那么该列是无法删除的。有外键约束的列也是无法删除的。比如要删除 Ssex 列（该列上有默认值 0），其执行脚本如下所示。

```
ALTER TABLE Student
DROP COLUMN Ssex
```

系统将会返回错误：

```
消息 5074，级别 16，状态 1，第 1 行
对象'DF_Student_Ssex' 依赖于 列'Ssex'。
消息 4922，级别 16，状态 9，第 1 行
由于一个或多个对象访问此列，ALTER TABLE DROP COLUMN Ssex 失败。
```

为了删除带有默认值或约束的列，必须要先使用 ALTER TABLE 删除约束。删除约束的语法格式为：

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name
```

下面就向 Student 表中添加 1 整形列 c3，该列的默认值为 10。在添加了带有默认值的列后再使用 ALTER TABLE 命令删除该列 c3，其脚本如代码 3.10 所示。

代码 3.10　添加删除带默认值的列

```
ALTER TABLE Student
ADD c3 int DEFAULT 10 --添加 c3 列，默认值为 10
GO
ALTER TABLE Student
DROP CONSTRAINT DF__Student__c3__0AD2A005 --删除默认值约束
GO
ALTER TABLE Student
DROP COLUMN c3 --删除列 c3
```

注意：SQL Server 对未命名的约束名称在每次创建时都不相同。如果读者运行代码 3.10 将会抛出异常，因为 DF__Student__c3__0AD2A005 是系统创建的名称，读者系统中创建的约束名和该约束名不相同。另外还需要注意的是，该名称中是使用连续的两个下划线，而不是一个下划线。

3. 使用ALTER TABLE修改列

SQL Server 中不但允许修改列的宽度，而且还允许修改列的数据类型。但是下列情况下数据类型不能更改。

- ❑ 大对象数据类型或 TIMESTAMP 类型。
- ❑ 索引的一部分。
- ❑ 主键或外键的一部分。
- ❑ 用于 CHECK 约束或 UNIQUE 约束中的。
- ❑ 有与之联系的默认值。
- ❑ 计算的或用在计算列中的。

当更改列的数据类型时，必须使已有数据与更改后的数据类型兼容。比如可以将整型更改为字符串类型。但是若需要将字符串类型更改为整型时，必须要保证表中该列的每行数据都能够转换为整型或为 NULL 值。使用 ALTER TABLE 修改列的语法格式如下：

```
ALTER TABLE table name
ALTER COLUMN column_name new_data_type
```

同样以前面使用的 Student 表为例。现在由于出现了特殊情况，需要将姓名字段加长到 20 个字符，那么修改脚本为：

```
ALTER TABLE Student
ALTER COLUMN Sname nvarchar(20)
```

4．使用ALTER TABLE修改主键

除了修改表的列长度和数据类型外，还可以使用 ALTER TABLE 来更改表的主键等。一个表只有一个主键。在向表中添加主键之前，必须先把以前的主键删除。删除主键的语法格式为：

```
ALTER TABLE table name
DROP CONSTRAINT constraint_name
```

在删除原有主键后便可以重新添加主键。其语法格式为：

```
ALTER TABLE table name
ADD CONSTRAINT constraint_name PRIMARY KEY(column_name [,…n])
```

以 Student 表为例。在增加了标识列 id 后，希望将主键更改为 id，而不是原来的 Sid，那么脚本如代码 3.11 所示。

代码 3.11　修改表的主键

```
ALTER TABLE Student
DROP CONSTRAINT PK Student --删除原来的主键
GO
ALTER TABLE Student
ADD CONSTRAINT PK_Student_id PRIMARY KEY (id) --增加新的主键
```

由于主键是表中数据的唯一标识，所以主键不能重复也不能为空。当使用 ALTER TABLE 修改表的主键时，新的主键列的数据定义必须是不为空的。SQL Server 中并没有提供直接修改主键的命令。若需要修改主键，只有通过删除原有主键添加新主键来完成。

3.2.6　使用 SSMS 修改表

与使用 T-SQL 修改表相比，SSMS 的可视化界面对表的修改操作则相对要简单得多。在 SSMS 中对表的修改步骤如下所述。

（1）使用 SSMS 登录到服务器并在对象资源管理器中选中需要修改的表。以前面使用的 Student 表为例，选中 Student 表。

（2）右击 Student 表，在弹出式菜单中选择“设计”选项，SSMS 将在主窗口新建一个选项卡并显示 Student 表的定义，如图 3.15 所示。显示表定义的界面与新建表时使用的窗口相似。

（3）若需要增加列或修改列可以直接在列定义表格中进行添加或修改。

（4）若需要删除列，只需要在要删除的列上右击，在弹出的快捷菜单中选择“删除列”选项即可。

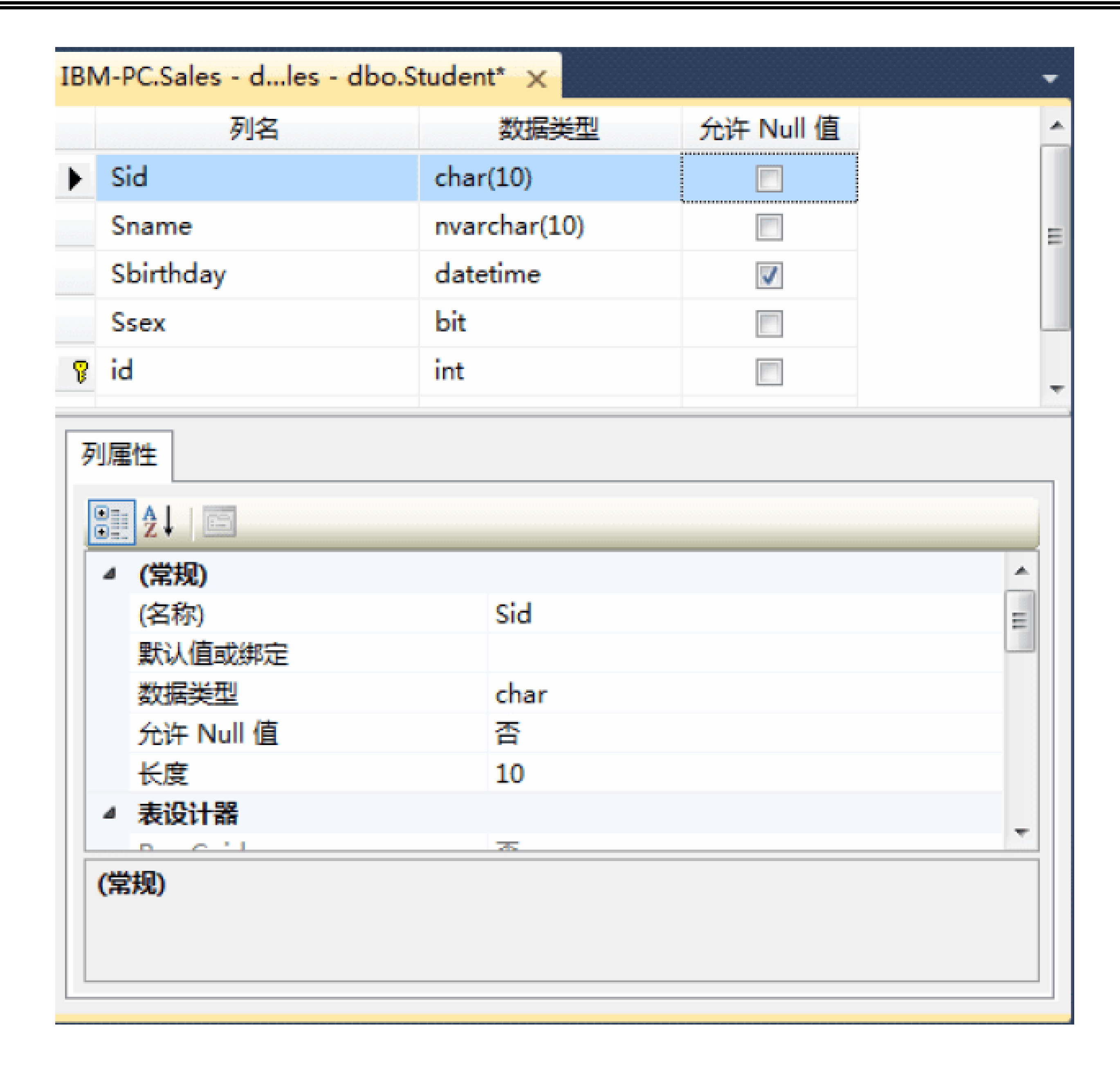


图 3.15　表的定义

技巧：删除列并不一定需要在图 3.15 那样的表定义窗口中删除。也可以直接在对象资源管理器中展开并选中需要删除的列，然后右击选择“删除”选项或者在选中要删除的列后直接使用快捷键 Delete。SSMS 将弹出删除确认窗口，在删除确认窗口中单击“确定”按钮即可删除列。

（5）若需要修改表的主键同创建表时设置主键类似。将需要设置为主键的列选中，然后单击工具栏中的“设为主键”按钮即可。

（6）单击工具栏中的“保存”按钮，SSMS 将把对表的修改应用到 DBMS 中。若修改成功，SSMS 的状态栏将显示“已保存项”。

（7）在弹出警告的情况下，若确定对表的修改，单击“是”按钮，SSMS 将忽略这些警告并将修改应用到 DBMS 中。

技巧：在 SSMS 中可以对表中字段的顺序进行调整，只需要选中要调整顺序的列左边的方块，然后通过拖动的方式便可实现列顺序的调整。

3.2.7　删除表

当某个表不再被使用时，希望将该表从数据库中删除，可以使用 T-SQL 中提供的 DROP TABLE 命令。

DROP TABLE 命令的语法格式为：

```
DROP TABLE table_name
```

同样以前面提到的 Student 表为例。若需要删除该表，则使用的命令为：

```
DROP TABLE Student
```

表一旦删除后将无法再找回。删除表时，表中的规则或默认值会失去绑定，还会自动删除与其相关的所有约束。删除后可以再重新创建一个与删除的表同名的表，但必须重新绑定适当的规则和默认值，添加所有必要的约束。

如果数据库中并不存在 Student 表，那么执行上面的删除表命令将会抛出异常。为了使 SQL 脚本能够多次运行，所以在执行表的删除时需要使用 OBJECT_ID 函数来判断某个表是否存在。那么删除 Student 表的脚本就应该改写为如代码 3.12 所示。

代码 3.12　删除表

```
IF  EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[Student]')
AND type in (N'U')) --判断表 Student 是否存在
    DROP TABLE [dbo].[Student]
```

DROP TABLE 命令提供了批量删除的功能。实现一次删除多张表，只需要将要删除的表使用“,”隔开即可。比如要删除表 A、B、C，那么执行的脚本为：

```
DROP TABLE A,B,C
```

注意： 有些用户为了方便直接将代码 3.12 简写为：IF(OBJECT_ID(N'Student') IS NOT NULL) DROP TABLE Student 这种写法是不正确的。这只判断了当前数据库中是否有数据库对象 Student，但并未判断 Student 对象是表。如果存在其他数据库对象命名为 Student，那么该语句就会抛出异常。

同样，在 SSMS 中删除表也是相当简单的。具体操作如下所述。

（1）使用 SSMS 连接上数据库并在对象资源管理器中选择需要删除的表。

（2）右击选择的表，并在弹出的快捷菜单中选择“删除”选项或者直接使用快捷键 Delete。系统将弹出对象删除窗口。

（3）在弹出的窗口中单击“确定”按钮，被选中的表即被删除。

虽然在对象资源管理器中并未提供多选的功能，但是使用 SSMS 仍然可以实现多表的批量删除。批量删除的操作方法如下所述。

（1）在 SSMS 的主区域中有一个特殊的选项卡叫做“对象资源管理器详细信息”。若读者没有看到该选项卡，可以选择“视图”菜单下的第二个选项“对象资源管理器详细信息”将该选项卡调出。

（2）在对象资源管理器中选中“表”，则详细信息选项卡中将列出该数据库下所有的表，如图 3.16 所示。



图 3.16　对象资源管理器详细信息

（3）按住 Ctrl 键多选需要删除的表。选定后右击，在弹出的快捷菜单中选择“删除”命令，如图 3.17 所示。

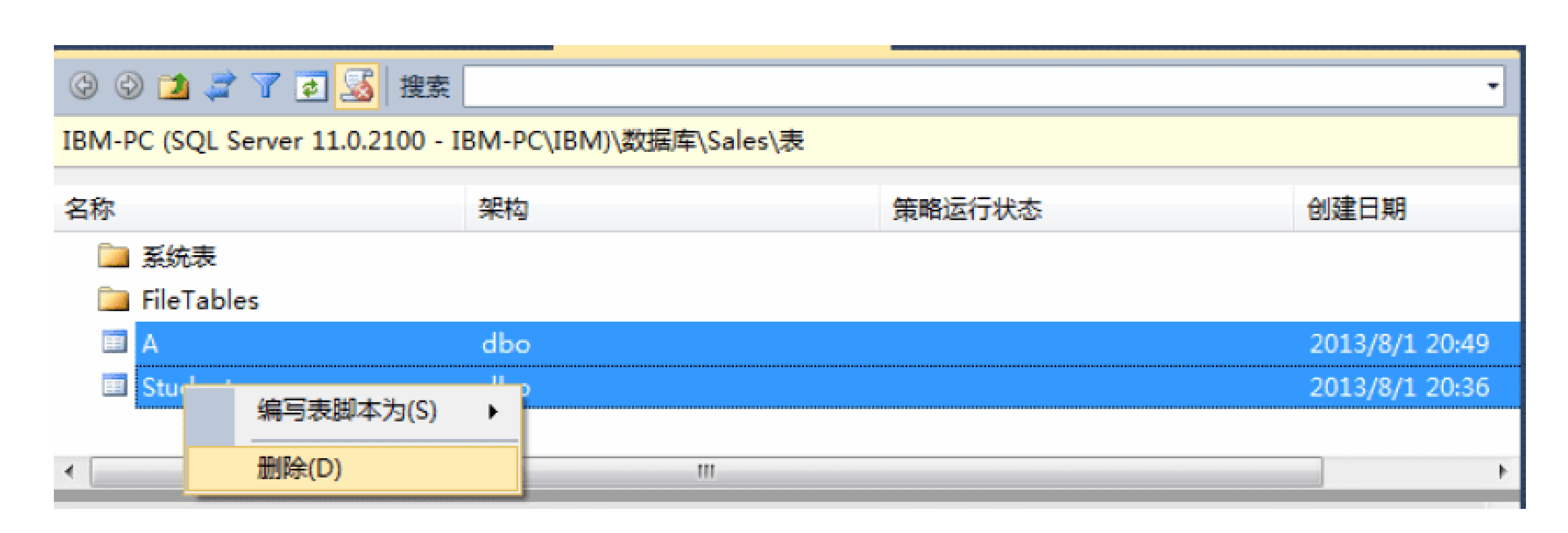


图 3.17　批量删除表

（4）在弹出的删除对象窗口中单击“确定”按钮，选中的表即被删除。

3.3　数据完整性

强制数据完整性可保证数据库中数据的质量。例如，如果一个数据库中存在 employee_id 值为 123 的雇员，则该数据库不应该允许其他雇员使用具有相同值的 ID。如果想将 employee_rating 列的值范围设定为 1～5，则数据库不应该接受值 6。如果表有一个存储雇员部门编号的 dept_id 列，则数据库应只允许接受有效的公司部门编号的值。

对表进行计划有两个重要步骤：标识列的有效值和确定如何强制列中数据的完整性。数据完整性的具体类别如下所述。

3.3.1　实体完整性

实体完整性将行定义为特定表的唯一实体。通过在表中建立唯一索引、UNIQUE 约束、主键（PRIMARY KEY，PK）约束或 IDENTITY 属性都可以实现实体完整性。

最简单的做法是，在表中定义了表的主键便可以实现实体完整性，主键用于唯一地标识出表中的每一行数据。也就是说，主键是不允许重复的。主键不一定是 1 列，可以由表的多个列组成联合主键。主键不允许为 NULL 值。每个表都应该有一个主键。

比如在一个学生管理系统中有一个学生表，那么学号就可以作为该表的主键，而为了唯一地区别一个班级，可以为每个班级设置一个 ID 作为主键。另外也可以使用毕业年份和班号作为联合主键。比如，2003 级 7 班便可以唯一地确定一个班级。

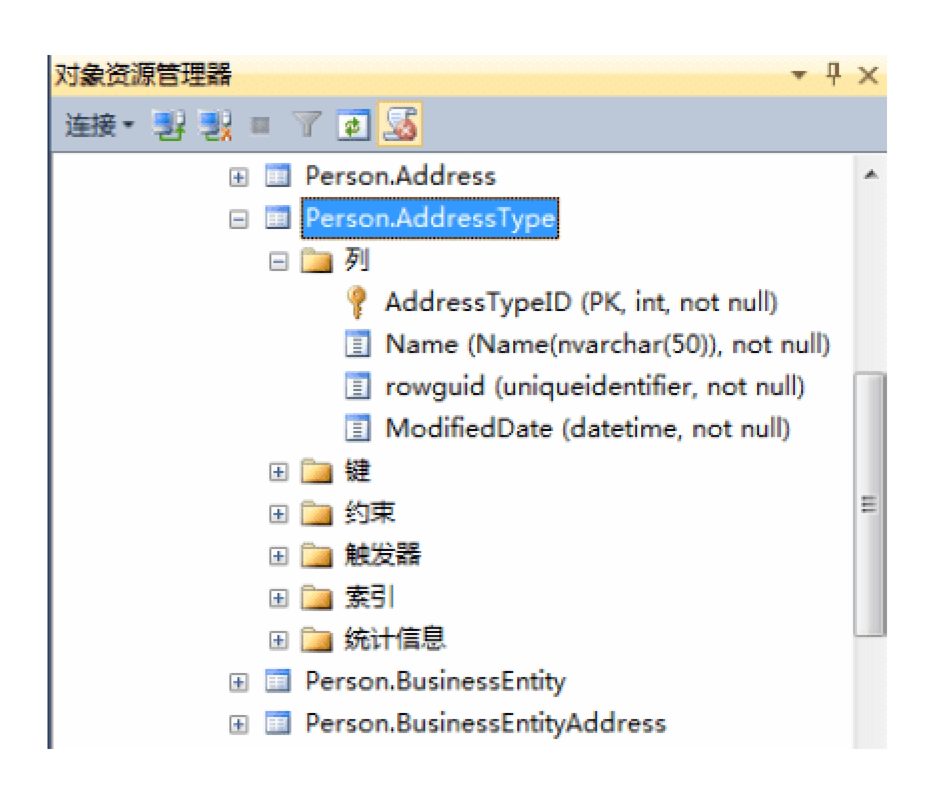


图 3.18　主键的显示

在 SSMS 中主键使用一个特殊的钥匙符号来表示，如图 3.18 所示的便是 AdventureWorks2012 数据库的 Person.AddressType 表。从图中的钥匙图案和旁边括号中的 PK 字符中可以看出，AddressTypeID 便是该表的主键。

除了使用实体实际存在的属性作为主键外，SQL Server 还提供了标识列（IDENTITY）作为表的主键。标识列在表中添加新行时，数据库引擎将为该列提供一个唯一的增量值。标识列通常与 PRIMARY KEY 约束一起用做表的唯一行标识符。

可以将 IDENTITY 属性分配给 tinyint、smallint、int、bigint、decimal(p,0)或 numeric(p,0)列。对于每个表，只能创建一个标识列。不能对标识列使用绑定默认值和 DEFAULT 约束，必须同时指定种子和增量，或者两者都不指定。如果二者都未指定，则取默认值（1，1）。

种子是向表中插入第一行数据时标识列生成的初始值。而增量就是在新插入一行数据时，标识列将在上一次生成的值上面增加一个增量值作为新的标识列值。

标识列是一直增长的（如果增量是负数，那么就是负向增长），与表中的实际数据量没有关系。在标识列为（1，1）的情况下，如果插入 100 行数据，然后再把这些数据全部删除，当再向表中插入数据时，标识列的值为 101 而不是 1。

3.3.2　域完整性

域完整性指特定列的项的有效性。可以通过使用数据类型对列限制类型，通过使用 CHECK 约束和规则限制格式，通过使用 UNIQUE 约束、CHECK 约束、DEFAULT 定义、NOT NULL 定义和规则可以限制列中可能值的范围。域完整性包括：

- 数据类型，对于实体的每一个属性都应该确定一种数据类型。比如一个班级的学生人数字段就应该使用 int 类型，那么该字段只能输入整数而不能输入小数或字符串。
- 是否为 NULL，对于实体的每一个属性都必须指定是否允许使用空值，如果不允许为空值那么这个字段必须输入值。比如学生表的学生姓名、性别等字段不能为空。
- CHECK 约束，指定应用于为列输入的所有值的布尔值（计算结果为 TRUE、FALSE 或未知）搜索条件。所有计算结果为 FALSE 的值均被拒绝。可以为每列指定多个 CHECK 约束。比如对班级的人数字段可以建立 CHECK 约束：人数>0，这样便杜绝了人数为 0 或负数的错误情况发生。
- UNIQUE 约束，用于强制实施列集中值的唯一性。但是与主键约束不同的是，UNIQUE 约束允许 NULL 值字段，而且一个表中可以建立多个 UNIQUE 约束。比如在学生表中使用学号作为主键，同时可以使用身份证号码作为 UNIQUE 约束。
- DEFAULT 定义，是为实体的属性定义一个默认值。在插入数据时若没有指定该列，则系统自动将该列的值设置为默认值。比如在学生表中可以设置性别字段默认为 1（一般性别字段设置为 BIT 类型，用 0 表示女，1 表示男）。

说明： 在 SQL Server 中还可以使用规则来维护域完整性，规则与 CHECK 约束相似，但是对于一个字段只允许一个规则，后续版本的 Microsoft SQL Server 将删除规则，所以不建议使用该功能。

3.3.3　引用完整性

引用完整性负责在插入、修改或删除记录时保持表之间已定义的关系。在 SQL Server 2012 中，引用完整性通过外键（FOREIGN KEY，FK）约束建立了表之间的引用关系。

引用完整性负责确保键值在所有表中一致。这里的一致性是指不引用不存在的值，如果一个键值发生更改，则整个数据库中，对该键值的所有引用要进行一致的更改。例如在学生管理系统中班级表和学生表之间存在引用关系，学生表中保存了学生所在班级的 ID。如果班级表中不存在 ID 为 100 的数据，则在学生表中也不能存在班级 ID 为 100 的学生数据。

所谓外键，就是当一个表的列被引用作为另一个表的主键值的列时，就在两表之间创建了链接，这个列就成为第二个表的外键。例如前面提到的班级表和学生表两个表，学生表中有字段班级 ID，该字段为班级表的主键。在设置了外键引用后只有在班级表中出现的班级 ID 才可能出现在学生表中。

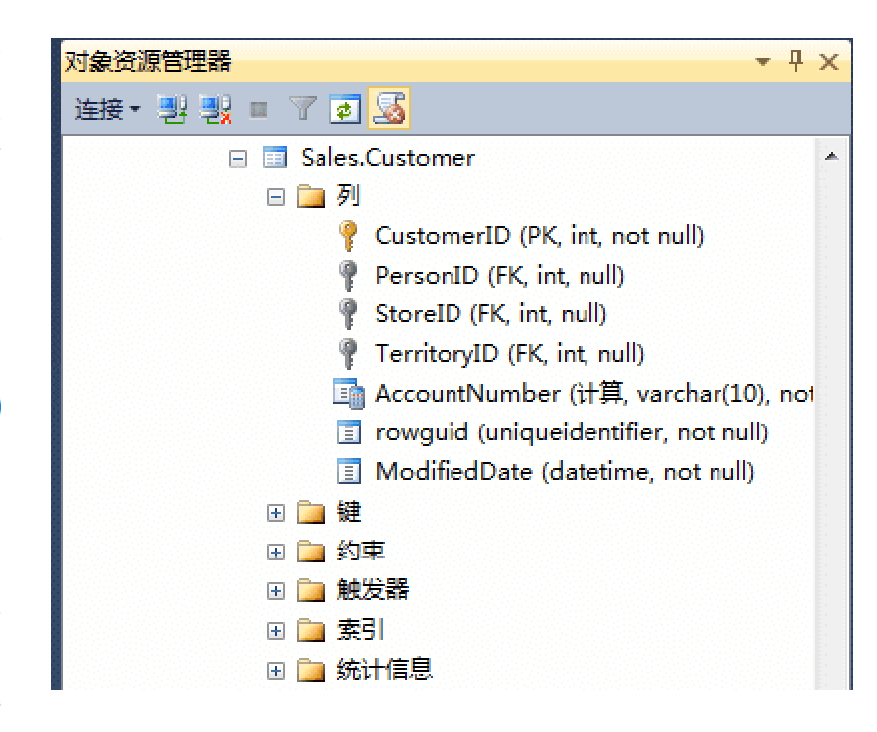


图 3.19　外键的显示

外键在 SSMS 中的表示与主键相似，不过外键是使用钥匙符号和 FK 来表示。图 3.19 中显示的便是 AdventureWorks2012 数据库的 Sales.Customer 表。其中，CustomerID 该表的主键。而从旁边括号中的 FK 可以看出，PersonID、StoreID 和 TerritoryID 是该表的外键。

3.3.4　用户定义完整性

用户定义完整性使用户可以定义不属于其他任何完整性类别的特定业务规则。所有完整性类别都支持用户定义完整性。例如在使用 CREATE TABLE 创建表时定义的所有列级约束和表级约束，另外其他存储过程以及触发器都可以定义完整性。这些特性将在后续章节进行讲解。

注意： 使用 SQL Server 2012 中的策略特性也可以在数据库中定义很多特殊功能的用户定义完整性。

3.4　约 束 操 作

约束是限制用户或应用程序输入表列数据的数据库对象。为了保证数据的完整性，约束起着关键性的作用。本节将主要讲解 SQL Server 2012 中的约束的使用。

3.4.1　约束简介

SQL 完整性约束通常简称为约束，可以分为 3 种类型。

- 与表相关的约束：在表定义中定义的一种约束，可以在表定义时作为列定义的一部分来创建约束，也可以在表定义时作为单独的一个元素来创建约束。第二种方式定义的约束可以用于一个和多个列，而第一种方式只能用于一个列。
- 断言：使用断言定义声明的一种约束。断言主要是通过 CHECK 约束方式实现，在断言中可以与一个或多个表进行关联。
- 域约束：在域定义中定义的一种约束。域约束也是通过 CHECK 约束方式实现，不过与断言不同的是域约束与在特定域中定义的任何列都相关。

在这 3 种约束中，最常用的约束是与表相关的约束。前面笔者已经提到，与表相关的约束可以分为表约束和列约束，表约束可以用于多个列，而列约束只能用于一个列。

在 SQL Server 2012 中总共有 6 种类型的约束，分别是 NOT NULL、UNIQUE、PRIMARY KEY、FOREIGN KEY、DEFAULT 和 CHECK。这些约束与 3 种约束类型的关系如图 3.20 所示。UNIQUE 约束和 PRIMARY KEY 约束被认为是唯一约束，而 FOREIGN KEY 约束被认为是参照约束。

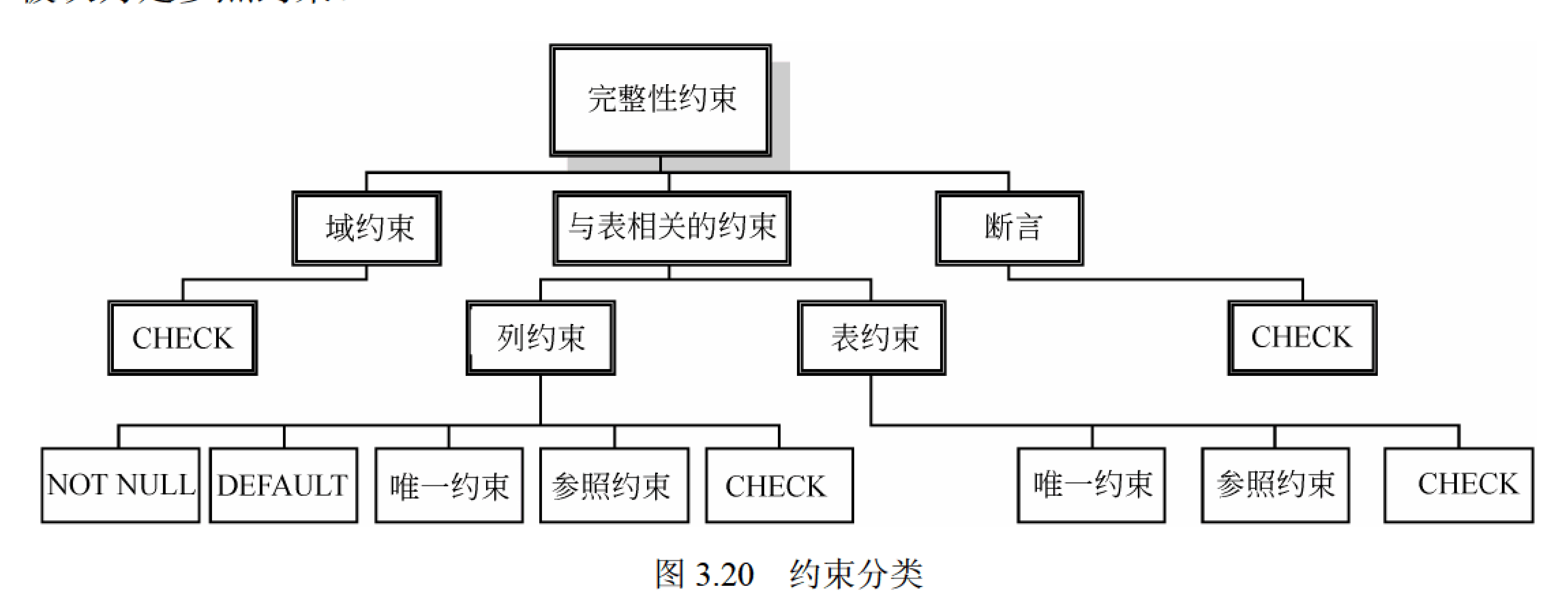


图 3.20　约束分类

3.4.2　NOT NULL 约束

在 3.2 节创建表时已经介绍过 NULL 表示未定义的值。NULL 并不等同于 0 或空白，也不能进行比较，而且 NULL 是创建表时字段的默认情况。NOT NULL 只能作为列约束来使用，而 NOT NULL 约束的创建也非常简单，其定义式为：

```
<column_name> {<data_type>|<domain>} NOT NULL
```

即只需要在创建或修改表时，将列名和数据类型后面跟“NOT NULL”即可。关于如何创建和修改表在前面已经进行了讲解，此处不再赘述。

注意：尽量避免允许空值，因为空值会使查询和更新变得更复杂，而且空值列与非空值列不能一起使用建立 PRIMARY KEY 约束。

3.4.3　DEFAULT 约束

在 SQL Server 中，每一条记录中的每列均必须有值，如果该值未定义则使用 NULL 表示。在实际应用中可能会有这种情况：必须向表中插入一行数据但不知道某一列的值，或该值尚不存在。如果列允许空值，就可以在插入数据时为该列指定为空值或不指定任何值。如果系统定义该列不允许为空，这种情况下就需要为列定义 DEFAULT 约束。

在执行插入数据操作时，如果在 SQL 语句中不指定某列，而且该列具有 DEFAULT 定义，则系统将会把 DEFAULT 定义中的默认值插入到该没有指定值的列中。

在创建表时，可以将 DEFAULT 约束作为表定义的一部分，也可以在每个列的定义中为该列定义 DEFAULT 约束。DEFAULT 约束虽然可以作为表定义的一部分进行申明，但是一个 DEFAULT 约束定义只能针对一个列，表中的每一列都可以包含一个 DEFAULT 约束。若要修改 DEFAULT 定义，必须首先删除现有的 DEFAULT 定义，然后再重新创建。

注意：DEFAULT 定义不能用于 timestamp 数据类型和 IDENTITY 或 ROWGUIDCOL 属性的数据列。

DEFAULT 定义除了指定常量外还可以使用函数。最常使用的一种函数 DEFAULT 定义就是在表中有 CreateTime 列 DATETIME 数据类型，用于记录一行数据插入的时间，可以将该列 DEFAULT 定义为 GETDATE()函数取得当前时间。这样用户就不需要专门去写该列，DEFAULT 定义就可以实现记录数据插入的时间。

另外一个比较常用的 DEFAULT 定义就是针对 uniqueidentifier 数据类型的数据列，这种数据类型用于记录 GUID，可用做主键。但是该类型并不像 int 类型那样有 IDENTITY 标识。幸好 SQL Server 中提供了 NEWID()函数用于产生 GUID，可以将 uniqueidentifier 数据类型的 DEFAULT 定义为 NEWID()。这样用户就不用去关心该列，系统在每次插入数据时都会自动生成一个 GUID。

注意：SQL Server 2012 中除了提供 NEWID()函数用于生成 GUID 外，还提供了另一个函数 NEWSEQUENTIALID()。该函数用于创建大于先前通过该函数创建的 GUID 的 GUID。该函数专门用于 uniqueidentifier 数据类型的 DEFAULT 约束，不能用于查询语句中。

关于 DEFAULT 定义的创建和修改已经在前面创建和修改表的章节中进行了介绍，这里就不再介绍了。

3.4.4　UNIQUE 约束

从图 3.20 中可以看到列约束和表约束都有唯一约束，而唯一约束又分为 UNIQUE 约束和 PRIMARY KEY 约束。本小节将主要讲解 UNIQUE 约束，关于 PRIMARY KEY 约束将在 3.4.5 节进行讲解。

UNIQUE 约束能够使一列或者一组列中只包含唯一的值。例如表 3.1 是某管理系统中

管理员表。管理员表的 AdminID 用于唯一地标识一个管理员，LoginName 是管理员登录名，Password 是经过系统加密后的密码。

表 3.1　管理员表

AdminID	LoginName	Password
1	admin	EC3F6D6E-B71E-46C0-8AA2-E4ADDF10EE86
2	zhangsan	6127F7AD-0F29-44F8-9D33-B76D1FD4AE38
3	yanwan	3FD8F6A4-FBAB-46D3-B16C-B668463C19D2
4	lihua	FA526E39-D0DC-4B87-9921-D0C2004F0696
5	hehuan	8754FD89-FFCF-4851-887A-637ACAA9BF14

虽然 AdminID 能够唯一地标识一个管理员，但是管理员的登录名 LoginName 也应该是唯一的，这样才能在管理员登录系统的时候标识出当前登录管理员。如果将 UNIQUE 约束应用于 LoginName 列，那么在增加或修改管理员数据时将无法插入重复的 LoginName。

在了解到 UNIQUE 约束的作用后，接下来将创建 UNIQUE 约束。前面讲到 UNIQUE 约束分为列约束和表约束。创建列约束时，只需要将 UNIQUE 关键字作为列定义的一部分添加进去即可，其语法如下：

```
<column name>(<data type>|<domain>) UINQUE
```

如前面提到的管理员表的创建 SQL 脚本如代码 3.13 所示。

代码 3.13　UNIQUE 列约束

```
CREATE TABLE [Admin]
(
    AdminID int PRIMARY KEY IDENTITY,
    LoginName varchar(50) NOT NULL UNIQUE, --指定该列为唯一约束的列
    Password varchar(50) NOT NULL
)
```

如果要把 UNIQUE 约束作为表约束来添加，那么必须把它作为表定义的表元素，语法如下：

```
CONSTRAINT <constraint name>
UNIQUE (<column name>[{,<column name>}...])
```

同样以管理员表为例，使用 UNIQUE 表约束方式创建表的 SQL 脚本，如代码 3.14 所示。

代码 3.14　UNIQUE 表约束

```
CREATE TABLE [Admin]
(
    AdminID int PRIMARY KEY IDENTITY,
    LoginName varchar(50) NOT NULL,
    Password varchar(50) NOT NULL,
    CONSTRAINT UNIQUE_LoginName UNIQUE(LoginName)
```

```
                                --以单独约束的方式指定唯一约束
)
```

UNIQUE 约束作为列约束比作为表约束应用要简单一些，但是列约束只能针对单独的列。如果要将多列作为一个 UNIQUE 约束，就只有使用表约束的方式。一个表中可以存在多个 UNIQUE 约束。

对于已经定义好的表也可以通过 ALTER TABLE 命令来增加 UNIQUE 约束。如对于已经创建好的没有 UNIQUE 约束的 Admin 表，增加对 LoginName 的 UNIQUE 约束的 SQL 脚本，如代码 3.15 所示。

代码 3.15　修改表添加 UNIQUE 约束

```
ALTER TABLE [Admin]
ADD CONSTRAINT UNIQUE_LoginName UNIQUE(LoginName) --修改表增加唯一约束
```

除了使用 T-SQL 语句为列或表添加 UNIQUE 约束外，还可以通过 SSMS 添加 UNIQUE 约束。具体操作如下所述。

（1）通过新建表或者设计表的方式进入 Admin 表的设计窗口。

（2）选择“表设计器”菜单中的“索引/键”选项，系统将出现“索引/键”对话框，如图 3.21 所示。此时 Admin 表还没有对 LoginName 创建 UNIQUE 约束，只是将 AdminID 作为主键。设计器中的 PK_Admin 是 Admin 表的主键，此处暂时不理会这个值。

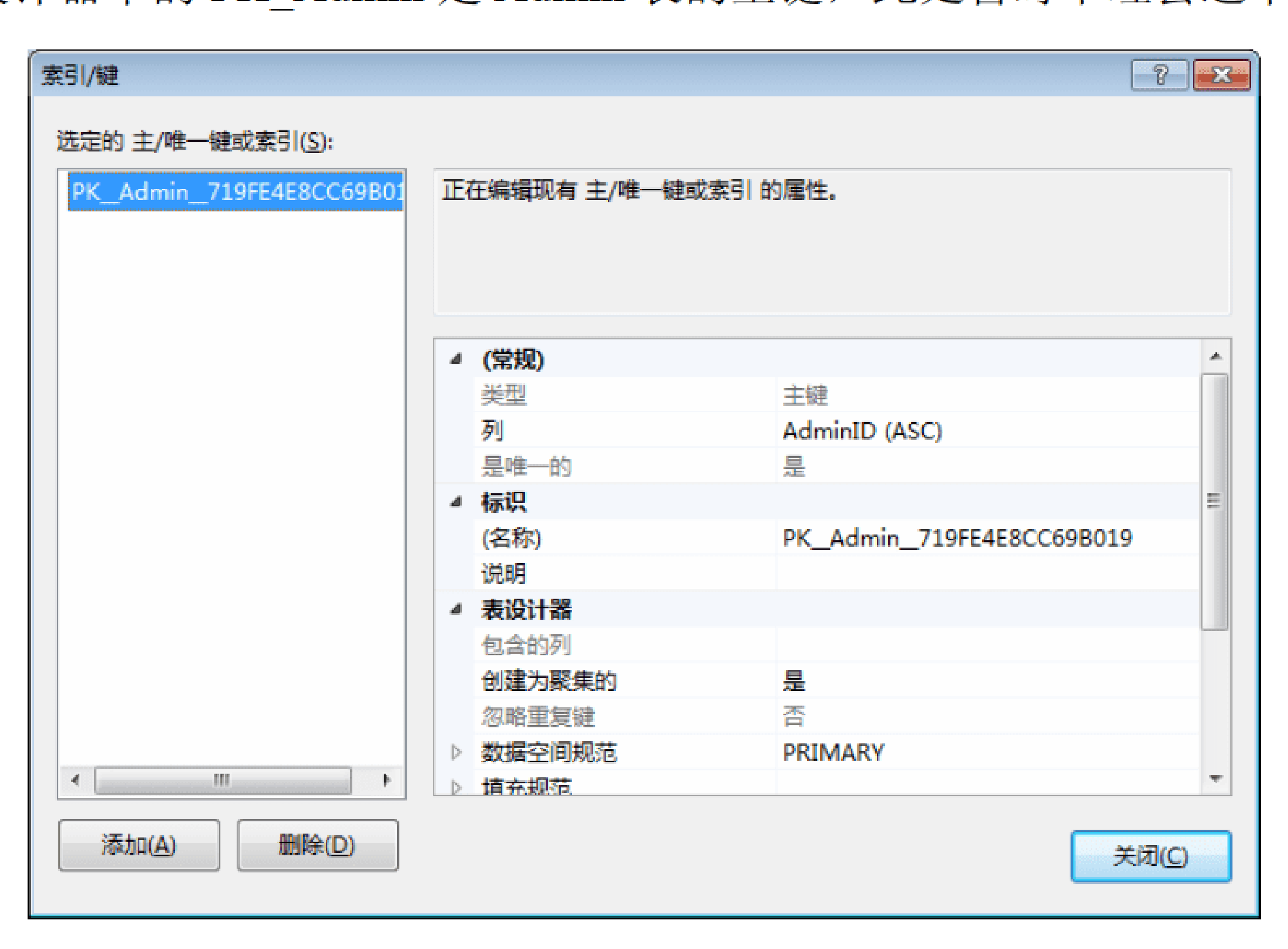


图 3.21　“索引/键”对话框

（3）在“索引/键”对话框中单击“添加”按钮，在左边列表中将会新建一行数据 IX_Admin。

（4）选中 IX_Admin，然后在右边的属性窗口中将“是唯一的”选项改为“是”，列修改为 LoginName，索引的名称修改为 UNIQUE_LoginName，如图 3.22 所示。

（5）单击“关闭”按钮，保存表设计，此时 UNIQUE 约束已经创建完成。在对象资源管理器中展开 Admin 表的“键”节点和“索引”节点，将可以看到创建的 UNIQUE 约束，

如图 3.23 所示。

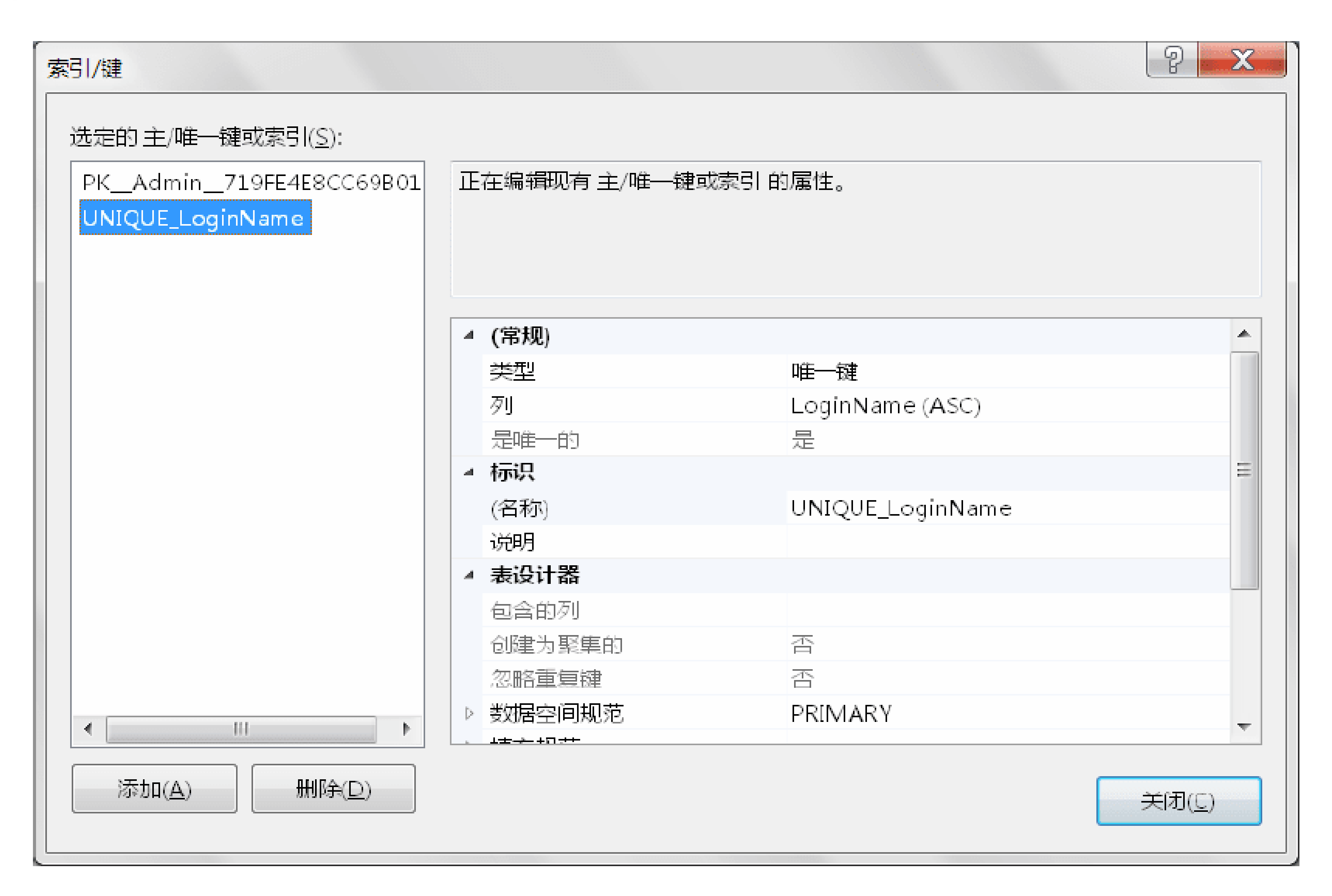


图 3.22　增加 UNIQUE 约束

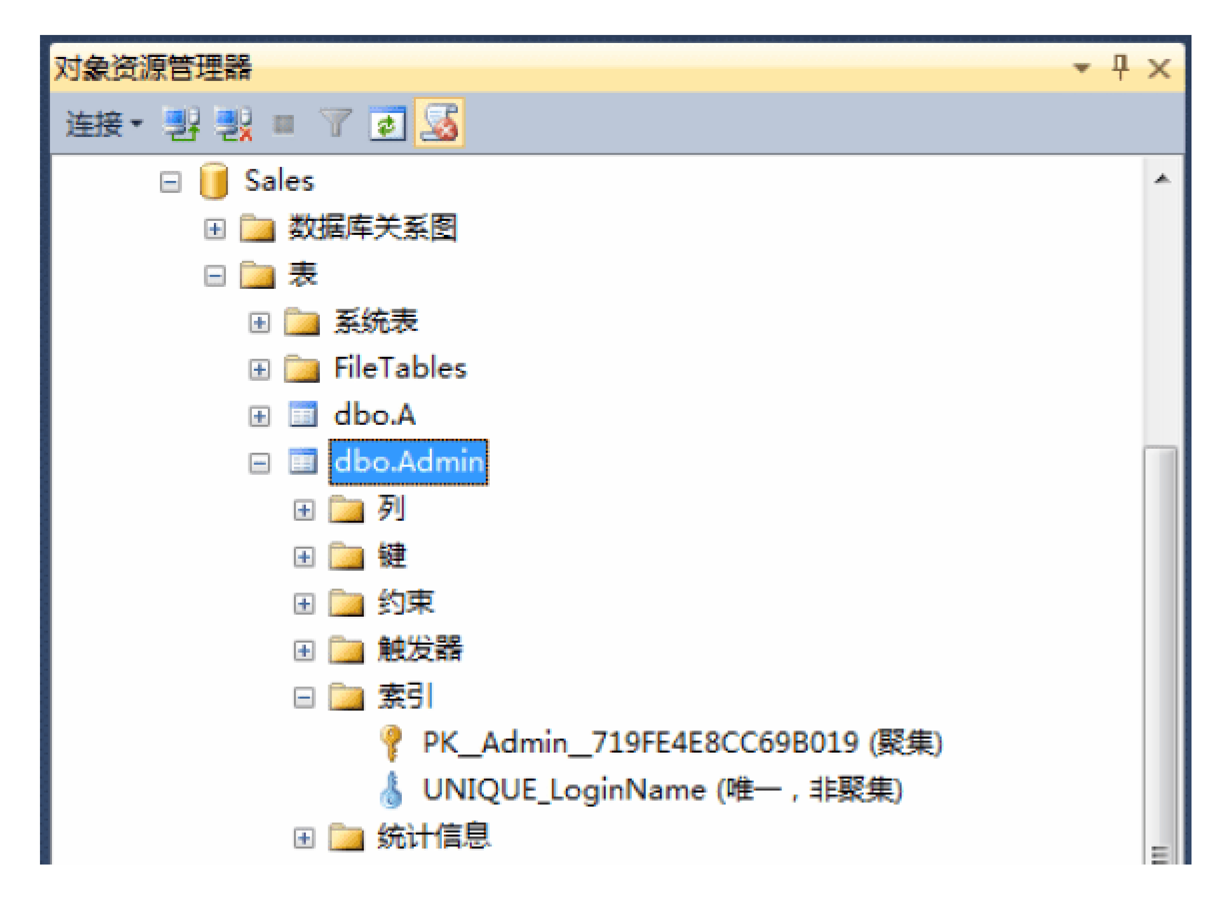


图 3.23　创建唯一约束后的对象资源管理器

3.4.5　PRIMARY KEY 主键约束

在前面已经提到，PRIMARY KEY 约束和 UNIQUE 约束都是 SQL 中的唯一性约束。这两种约束只许可指定列中的唯一值，被指定的列可以是单独一列也可以是多列。但是 PRIMARY KEY 约束和 UNIQUE 约束的主要区别如下：

- ❑ 利用 PRIMARY KEY 约束定义的列不能包含 NULL 值，而 UNIQUE 约束指定的列却可以为 NULL 值。
- ❑ 对于每一个表只能指定一个 PRIMARY KEY 约束，但却可以指定多个 UNIQUE 约束。

要理解这些限制的原因，必须了解关系数据库的知识。关系数据库是基于数据的“相关”性而创建的。因此在关系数据库中，对于大多数表来说每一行拥有一个唯一标识是十

分严格的。通过唯一标识允许将该表中的记录与其他表中的记录精确相关，这样就形成了两个表的关系。

主键就是表中数据的唯一标识，它必须包含唯一值，所以不能为 NULL 值。而为了能够对其他表进行“相关”，所以主键在一个表中只能有一个。

要定义主键就必须使用 PRIMARY KEY 约束来指定哪个列或哪些列将作为表的主键。定义 PRIMARY KEY 约束的方式与定义 UNIQUE 约束的方式基本相同。如需要将 PRIMARY KEY 约束添加到列定义的语法为：

```
<column name>(<data type>|<domain>) PRIMARY KEY
```

以前面提到的 Admin 表为例。AdminID 作为主键，在不考虑 LoginName 的 UNIQUE 约束的情况下，Admin 表的定义如代码 3.16 所示。

代码 3.16　PRIMARY KEY 列约束

```
CREATE TABLE [Admin]
(
    AdminID int PRIMARY KEY IDENTITY, --设置该列为主键
    LoginName varchar(50) NOT NULL,
    Password varchar(50) NOT NULL
)
```

说明： PRIMARY KEY 列定义与 NOT NULL PRIMARY KEY 列定义的含义是相同的。由于主键不能为 NULL，所以只要定义为 PRIMARY KEY 的列必然是不为 NULL 的。

对于 PRIMARY KEY 表约束，其语法如下：

```
CONSTRAINT <constraint name>
PRIMARY KEY (<column name>[{,<column name>}…])
```

在创建表时 PRIMARY KEY 约束可以作为表元素直接写在表中。同样以 Admin 表为例，创建 PRIMARY KEY 表约束的 SQL 脚本如代码 3.17 所示。

代码 3.17　PRIMARY KEY 表约束

```
CREATE TABLE [Admin]
(
    AdminID int IDENTITY,
    LoginName varchar(50) NOT NULL,
    Password varchar(50) NOT NULL,
    CONSTRAINT PK_AdminID PRIMARY KEY(AdminID) --单独以约束方式设置主键
)
```

如果 Admin 表已经存在，而且已经将 AdminID 设置为了主键，但是现在想把 LoginName 变为该表的主键，那么可以先将约束 PK_AdminID 删除，然后再添加新的约束 PK_LoginName。其 SQL 脚本如代码 3.18 所示。

代码 3.18　修改表的主键

```
ALTER TABLE Admin
DROP CONSTRAINT PK_Admin --删除原有主键约束
```

```
GO
ALTER TABLE Admin
ADD CONSTRAINT PK_LoginName --增加新的主键约束
PRIMARY KEY (LoginName)
```

同样地，SSMS 也提供了可视化的界面来对表的主键进行设置。以代码 3.18 修改后的 Admin 表为例，要在 SSMS 中修改该表的主键为 AdminID，具体操作步骤如下所述。

（1）在 SSMS 的对象资源管理器中右击 Admin 表。在弹出的快捷菜单中选择“设计”选项，系统将进入 Admin 表的设计窗口，如图 3.24 所示。

（2）LoginName 左边的方格中有一个钥匙图标表明当前表的主键是 LoginName。选择 AdminID，单击“表设计器”菜单的“设置主键”选项或工具栏上的“设置主键”按钮，钥匙图标将出现在 AdminID 左边的方格中，表明将 AdminID 设置为主键。

（3）保存并关闭表设计窗口，Admin 表的主键已经修改为 AdminID。

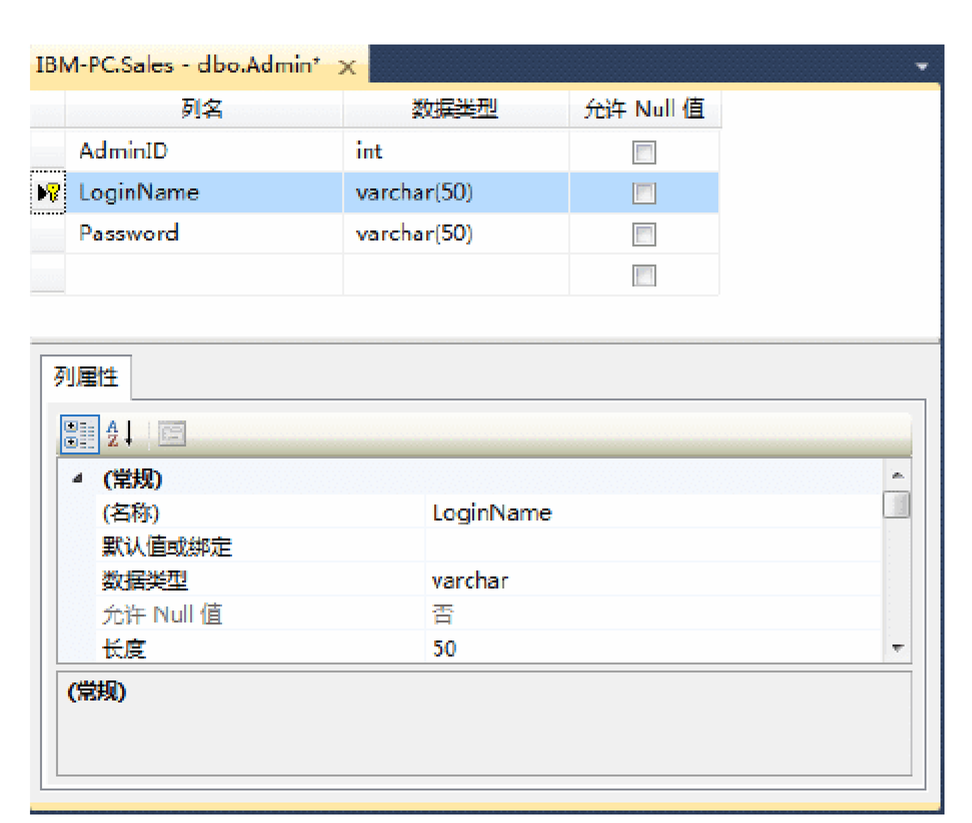


图 3.24　Admin 表设计窗口

3.4.6　FOREIGN KEY 外键约束

前面主要讨论的是针对单个表约束，但是 FOREIGN KEY 约束却不同。因为该约束涉及的是一个表中的数据与另一个表中的数据的关系，这就是将其称之为参照约束的原因。

1．添加FOREIGN KEY约束

外键约束是确保数据完整性并显示表之间关系的一种方法。在一张表上添加了外键，实际上就是在创建定义外键的表（称为参照表或被引用表）和外键引用的表（称为引用表）之间的依赖关系。添加完 FOREIGN KEY 约束后，插入到参照表的任何数据的外键字段要么在引用表中有记录，要么其值就是 NULL，不能出现引用的值在另一个表中不存在的情况。

在创建外键约束时必须遵循以下几个原则：

- 被引用列必须是引用表的候选键，最常见的情况是将参照表的主键作为被引用列。
- 和主键约束相同，外键约束也分为列约束和表约束两种。同样是列约束只能包含一列，表约束可以包含一列或多列。
- 无论是引用多个列还是单个列，引用表外键包含的列都必须与参照表中对应的列数相同，并且对应列的数据类型也必须相同。但是引用列的名称可以与参照表中对应的列不同。
- 在定义外键约束时如果没有指定参照表中的参照列，那么参照表的主键就是参照列。

在了解了以上几个原则后，下面使用 SQL 语句来创建外键。

如果将外键约束作为列约束添加，那么必须把该约束添加到列定义中，其语法格式为：

```
<column name>(<data type>|<domain>) [NOT NULL]
REFERENCES <referenced table>[(<referenced columns>)]
```

以一个企业的组织结构数据库为例，现有部门表 Department（包含主键 DepartmentID 和部门名 DepartmentName）和员工表 Employee（包含主键 EmployeeID、员工姓名 EmployeeName 和所在部门 DepartmentID）。在没有建立 FOREIGN KEY 约束的情况下，员工所在部门的 ID 将可以是任意值，即使这个值不在部门表中。现需要在 Employee 表的 DepartmentID 字段上创建 FOREIGN KEY 约束关系到 Department 表的主键 DepartmentID，其 SQL 脚本如代码 3.19 所示。

代码 3.19　创建 FOREIGN KEY 列约束

```
CREATE TABLE Department  --创建部门表
(
    DepartmentID int IDENTITY PRIMARY KEY,
    DepartmentName nvarchar(20) NOT NULL
)
GO
CREATE TABLE Employee  --创建员工表
(
    EmployeeID int IDENTITY PRIMARY KEY,
    EmployeeName nvarchar(10) NOT NULL,
    DepartmentID int NOT NULL
    FOREIGN KEY REFERENCES  Department(DepartmentID)  --外键约束
)
```

如果要将 FOREIGN KEY 约束作为表约束来添加，那么必须将其作为表定义中的表元素。其语法格式为：

```
CONSTRAINT <constraint name>
FOREIGN KEY (<referencing column>[{,< referencing column >}…])
REFERENCES < referenced table > [<referenced columns>]
```

同样以部门员工表为例，使用表约束的方式创建 FOREIGN KEY 的 SQL 脚本，如代码 3.20 所示。

代码 3.20　创建 FOREIGN KEY 表约束

```
CREATE TABLE Department
(
    DepartmentID int IDENTITY PRIMARY KEY,
    DepartmentName nvarchar(20) NOT NULL
)
GO
CREATE TABLE Employee
(
    EmployeeID int IDENTITY PRIMARY KEY,
    EmployeeName nvarchar(10) NOT NULL,
    DepartmentID int NOT NULL,
    CONSTRAINT FK_Employee_Department  --单独以约束的方式定义外键约束
        FOREIGN KEY(DepartmentID)
        REFERENCES  Department(DepartmentID)
)
```

注意：在创建 FOREIGN KEY 约束之前，参照表必须已经存在，而且参照列必须在参照表只有 UNIQUE 约束或 PRIMARY KEY 约束时才能成功创建 FOREIGN KEY 约束。

对于已经创建好的两个表，可以通过 ALTER TABLE 命令将外键约束添加到表中。如对于没有外键约束的表 Department 和 Employee，增加两者之间的外键约束的 SQL 脚本，如代码 3.21 所示。

代码 3.21　增加 FOREIGN KEY 约束

```
ALTER TABLE Employee
ADD CONSTRAINT FK Employee Department
FOREIGN KEY(DepartmentID)
REFERENCES Department  --不指定参照列则参照列就是Department的主键DepartmentID
```

注意：如果 B 表通过外键引用了 A 表，那么 SQL Server 是不允许删除被引用的 A 表的。若要删除 A 表，则必须删除所有对 A 表的外键引用。

2. 自参照表

在外键约束中有一种比较特殊的情况，那就是参照表和引用表是同一张表，这种表就叫做自参照表。自参照表是一种常规的应用，常用于建立树结构。比如以部门表 Department 为例，由于存在一级部门、二级部门、三级部门的情况，所以需要增加一个字段 ParentID，用于表示当前部门的上级部门。为了保证数据的完整性，所以需要在 Department 表上增加 FOREIGN KEY 约束，将 ParentID 作为引用列而将 DepartmentID 作为参照列。

可能很多读者已经意识到了，那么一级部门怎么办？一级部门没有上级部门，那么 ParentID 又应该是什么？对于这个问题一般有两种解决办法。其中，一种是将 ParentID 设置为允许 NULL 值。对于一级部门，则其 ParentID 为 NULL。该表的 SQL 脚本如代码 3.22 所示。

代码 3.22　创建自参照表

```
CREATE TABLE Department2
(
    DepartmentID int IDENTITY PRIMARY KEY,
    DepartmentName nvarchar(20) NOT NULL,
    ParentID int NULL REFERENCES Department2(DepartmentID) --指定自参照外键
                                                           约束
)
```

另一种办法是在 Department 未设置自参照时在表中增加一行特殊的数据作为基本行（比如，DepartmentID 为 1，DepartmentName 为“虚拟部门”，ParentID 为 1），然后再将 Department 表设置为自参照，所有一级部门的 ParentID 设置为 1。使用这个办法的 SQL 脚本如代码 3.23 所示。

代码 3.23　创建带基础行的自参照表

```
CREATE TABLE Department3  --创建无外键约束的表
(
    DepartmentID int IDENTITY PRIMARY KEY,
    DepartmentName nvarchar(20) NOT NULL,
    ParentID int NOT NULL
)
GO
INSERT INTO Department3(DepartmentName,ParentID) --添加基础行
VALUES('虚拟部门',1)
GO
ALTER TABLE Department3  --添加外键约束
ADD CONSTRAINT FK_Department_Department
FOREIGN KEY (ParentID) REFERENCES Department3(DepartmentID)
```

3．级联（Cascading）更新和删除

外键除了限制引用列中的值之外，还限制了参照列的操作。外键是双向的，无论用户在参照表做了什么，外键都将检查引用表，避免出现不完整的记录。

对于 SQL Server 而言，默认情况下如果参照表中的某行数据被引用，那么将不允许对该行删除。但是若希望在更新和删除参照表数据的同时，自动删除引用表中对应的行或者将对应行的引用列设置为 NULL，那么将用到级联更新和删除。级联更新删除是 FOREIGN KEY 约束语法中的一部分，其语法格式如下：

```
[ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
[ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
```

其中，ON DELETE 表示级联删除操作，ON UPDATE 表示级联更新操作。NO ACTION 是 SQL Server 的默认值，表示不执行操作。CASCADE 层叠操作表示级联自动删除引用表相关数据。SET NULL 表示将引用表中的相关引用列设置为 NULL。如果引用列定义了 NOT NULL 约束则不能使用该选项。SET DEFAULT 表示将引用列中的数据设置为默认值。如果引用列未定义 DEFAULT 值，则不能使用该选项。

以前面提到的部门人员表为例，若需要将外键设置更新时不执行操作，删除时进行级联删除，则创建这两个表的 SQL 脚本如代码 3.24 所示。

代码 3.24　定义级联删除

```
CREATE TABLE Department
(
    DepartmentID int IDENTITY PRIMARY KEY,
    DepartmentName nvarchar(20) NOT NULL
)
GO
CREATE TABLE Employee
(
    EmployeeID int IDENTITY PRIMARY KEY,
    EmployeeName nvarchar(10) NOT NULL,
    DepartmentID int NOT NULL,
    CONSTRAINT FK_Employee_Department  --定义外键约束
        FOREIGN KEY(DepartmentID)
        REFERENCES  Department(DepartmentID)
```

```
        ON UPDATE NO ACTION --更新时不执行操作
        ON DELETE CASCADE --删除时进行级联删除
)
```

使用代码 3.24 生成的表，由于定义了级联删除，如果在 Department 表上执行 DELETE 操作，将会同时删除 Department 表中的数据和 Employee 中的相关数据。也就是说，执行了一条 DELETE 命令，两张表的记录都被删除了，一条命令操作了多个表。

那么如果还有一个表 A 与 Employee 有外键关系并建立了级联删除会怎么样？答案是在删除 Department 表记录的同时也会删除 A 表中的相关记录。如果还有 B 表与 A 表建立级联删除的外键关系，B 表中的相关记录也会被删除。级联后影响的深度可以是无限的。

正是由于级联的这一特性，使得数据库操作人员不易意识到 DELETE 和 UPDATE 语句在数据库中执行的操作。由于这个原因，所以笔者建议在数据库中不要建立太多的级联操作，尤其是实体关系是否复杂的系统更是如此。

技巧： 既要实现引用完整性而又想避免级联删除带来的问题时，就是只使用 NO ACTION 的外键约束，而将删除操作改为逻辑删除，即在表中设置一个删除标记字段。对数据的删除操作只是修改这个标记而已，不用真正删除数据。

4．使用SSMS操作外键

使用 SSMS 的可视化界面对外键进行操作将十分简单。同样以部门员工表为例，先建立两个表 Department 和 Employee，两表之间相互独立，未建立 FOREIGN KEY 约束。现使用 SSMS 为 Employee 表建立外键约束的操作如下所述。

（1）右击 Employee 表，在弹出的快捷菜单中选择“设计”选项，进入 Employee 表的设计窗口。

（2）选择“表设计器”菜单下的“关系”命令，系统弹出“外键关系”对话框，如图 3.25 所示。

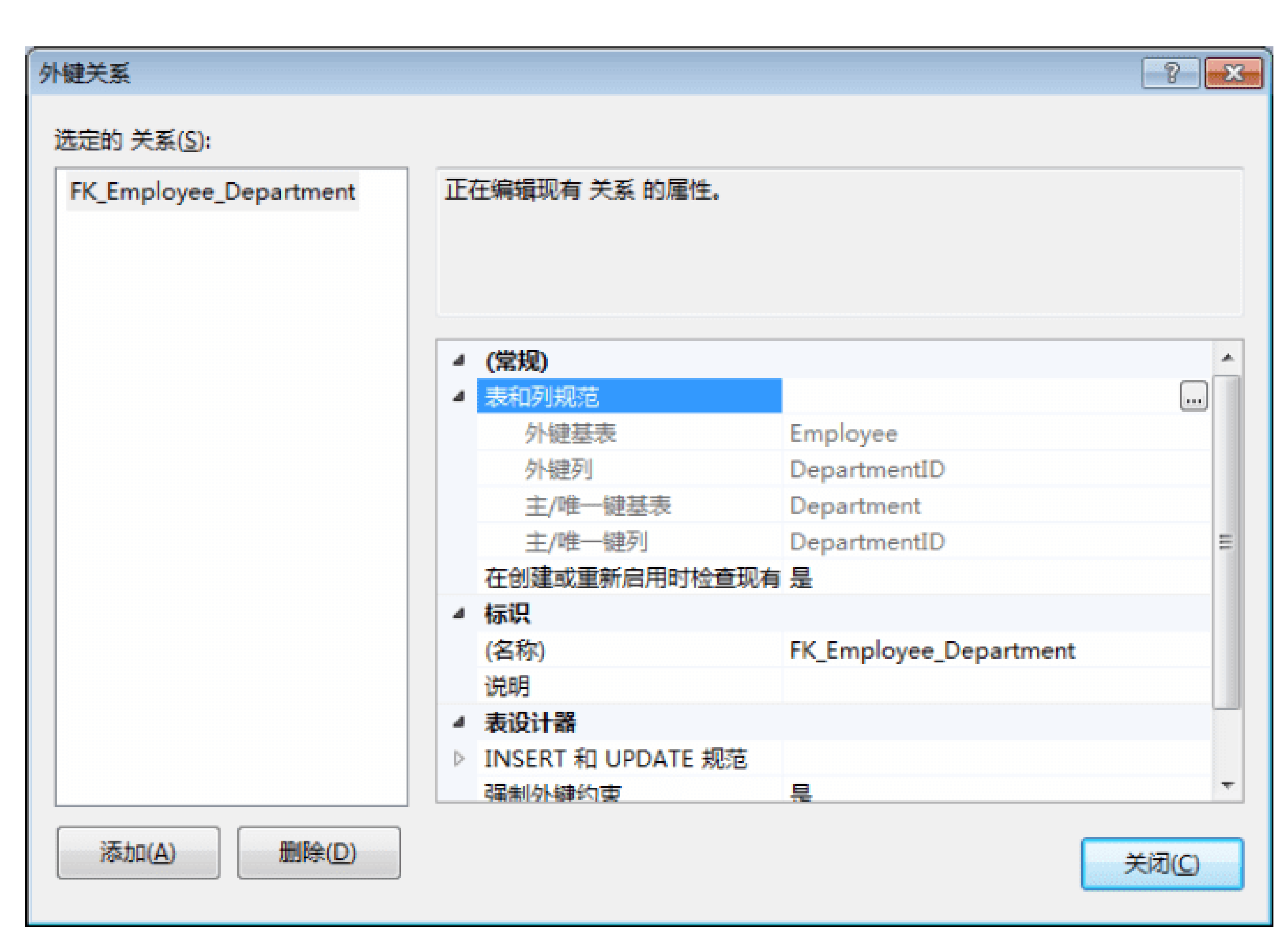


图 3.25　“外键关系”对话框

（3）单击“添加”按钮，系统将在左边“选定的关系”区域中新建一个关系 FK_Employee_Department。

（4）在右侧的属性窗口中选中“表和列规范”选项，并单击旁边的省略号按钮，系统将弹出“表和列”对话框。

（5）将主键表修改为 Department 并选择 DepartmentID 作为参照列，将外键表中的 DepartmentID 选为引用列，如图 3.26 所示。

（6）单击“确定”按钮，回到外键关系窗口。

（7）展开“INSERT 和 UPDATE 规范”，将“删除操作”设置为“级联”，即级联删除，如图 3.27 所示。

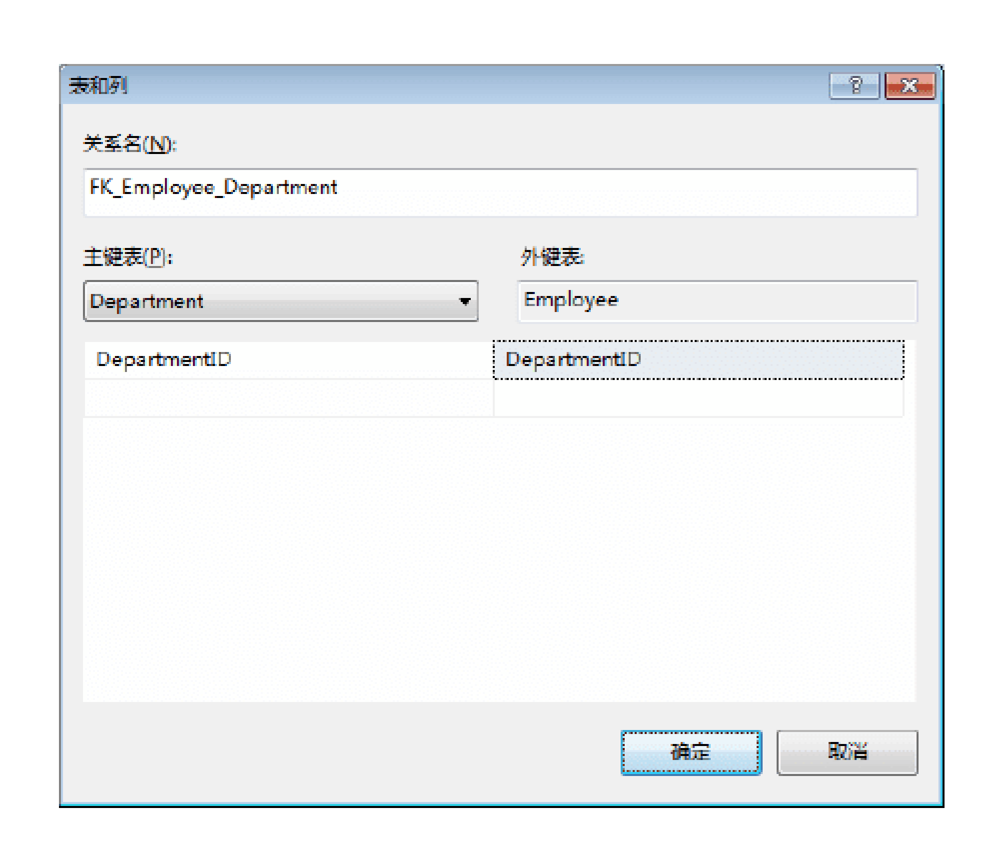


图 3.26　“表和列”对话框

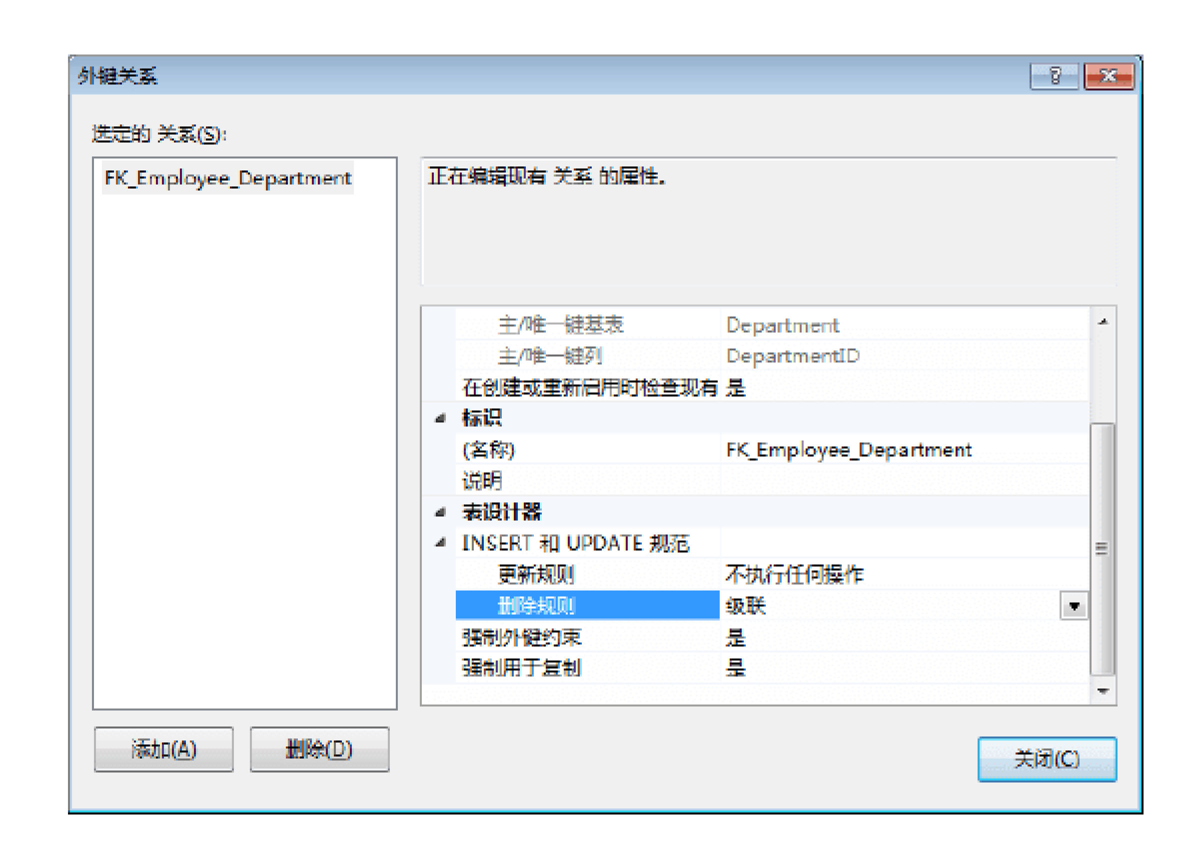


图 3.27　设置外键删除规则

（8）单击“关闭”按钮，回到表设计窗口，单击工具栏“保存”按钮提交对表的修改。

3.4.7　CHECK 约束

CHECK 约束的优点在于它不局限于特定的列，是所有约束中最灵活的约束，同时也是最复杂的约束。CHECK 约束可以被定义为表约束、列约束、域约束或者断言中，其可以指定包含在列中的值，比如定义范围（如 0～100），枚举（如 A、B、C、D）或者其他许可值条件。要创建 CHECK 列约束，应该在列的定义中使用以下语法：

```
<column name>(<data type>|<domain>) CHECK(<search condition>)
```

例如要创建一个用户表 User，其中 Age 字段要求在 0～120 之间，那么创建该表的 SQL 脚本如代码 3.25 所示。

代码 3.25　创建 CHECK 列约束

```
CREATE TABLE [User]
(
    UserID int IDENTITY PRIMARY KEY,
    UserName nvarchar(10) NOT NULL,
    Age int CHECK(Age>=0 AND Age<=120) --指定该列的 CHECK 约束
)
```

添加 CHECK 约束后，在向 User 表中插入数据或更改数据时，都必须保证 Age 满足 CHECK 约束条件，否则插入或更新将失败。

要创建 CHECK 表约束，在表定义中的语法为：

```
[CONSTRAINT <constraint name>] CHECK(<search condition>)
```

例如，要创建一个病人表 Sick，其中血型字段 BloodType 用于记录病人的血型，那么创建该表的 SQL 脚本如代码 3.26 所示。

代码 3.26　创建 CHECK 表约束

```
CREATE TABLE Sick
(
    SickID int IDENTITY PRIMARY KEY,
    SickName nvarchar(5) NOT NULL,
    BloodType varchar(2) NOT NULL,
    CONSTRAINT CK_BloodType CHECK(BloodType IN ('A','B','AB','O'))
    --以单独约束方式定义 CHECK 约束
)
```

创建的 Sick 表中使用 CHECK 约束规定了 BloodType 的值必须是 A、B、AB、O 这 4 种血型中的一种。如果插入一条记录中的 BloodType 为 C 型，由于是无效数据，不满足 CHECK 约束，插入将失败。

CHECK 约束不仅可以对一列的输入进行检查，也可以对多列进行约束。例如有个会议记录表 Meeting，用于记录会议的主题 Topic、开会时间 StartTime 和结束时间 EndTime。若需要建立 CHECK 约束，要求 StartTime 必须早于 EndTime，则创建该表的 SQL 脚本如代码 3.27 所示。

代码 3.27　创建多列比较 CHECK 约束

```
CREATE TABLE Meeting
(
    MID int IDENTITY PRIMARY KEY,
    Topic nvarchar(50) NOT NULL,
    StartTime datetime NOT NULL,
    EndTime datetime NOT NULL,
    CONSTRAINT CK_Time CHECK(StartTime<EndTime) --多个列之间的 CHECK 约束
)
```

这里实际只谈到了 CHECK 约束的基础知识，实际上 CHECK 约束的功能远不止这些。几乎所有 WHERE 从句完成的任务都可以放置到这种约束下完成。

使用 SSMS 操作 CHECK 约束的方式与操作 UNIQUE 约束十分相似。以前面提到的病人表 Sick 为例，先在数据库中创建未添加 CHECK 约束的 Sick 表。使用 SSMS 为 Sick 表添加 CHECK 约束的操作如下所述。

（1）右击 Sick 表，在弹出的快捷菜单中选择“设计”选项，进入 Sick 表的设计窗口。

（2）选择“表设计器”菜单下的“CHECK 约束”选项，系统弹出“CHECK 关系”对话框，如图 3.28 所示。

（3）单击“添加”按钮，系统将在左边列表中新建一行数据 CK_Sick。

（4）在“表达式”文本框中输入 CHECK 表达式：BloodType IN ('A','B','AB','O')，如图 3.29 所示。

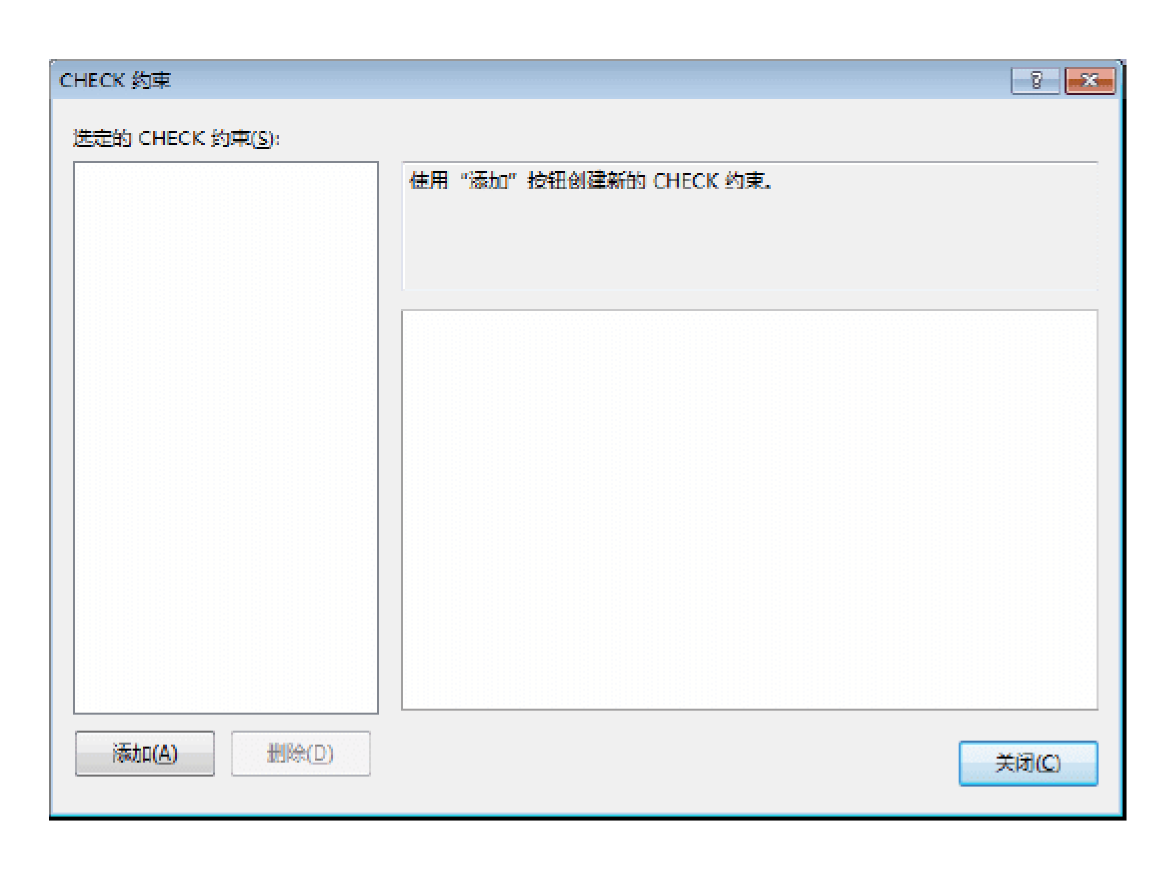


图 3.28　“CHECK 约束”对话框

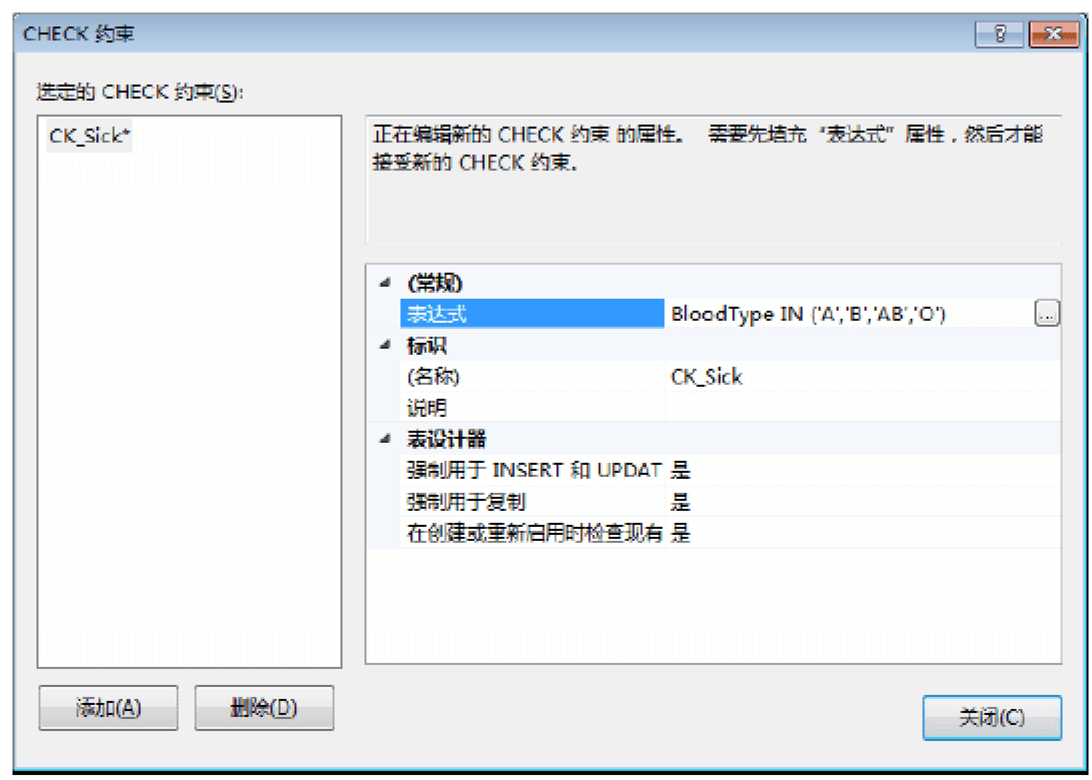


图 3.29　添加 CHECK 约束

（5）单击“关闭”按钮，回到表设计器窗口。

（6）单击工具栏“保存”按钮，提交对表的修改。

3.4.8　规则

规则（Rules）存在的时间比 CHECK 约束存在的时间要长，规则是早期 SQL Server 的概念。为了保证向后兼容性，SQL Server 2012 保留了规则，但是不能保证在以后的版本中仍将继续保留这一特性，所以笔者建议在使用中都使用 CHECK 约束来代替规则。

规则与 CHECK 约束十分相似，最主要的区别就是规则一次只工作在一列上。像代码 3.27 中的 StartTime<EndTime 约束这种对多列的约束是无法用规则实现的。规则的创建语法如下：

```
CREATE RULE [ schema name . ] rule name
AS condition_expression
```

condition_expression 定义规则的条件。规则中包括一个变量。规则可以是 WHERE 子句中任何有效的表达式，并且可以包括诸如算术运算符、关系运算符和谓词（如 IN、LIKE、BETWEEN）这样的元素。规则不能引用列或其他数据库对象，可以包括不引用数据库对象的内置函数，不能使用用户定义函数。

规则创建后并不能直接使用。因为创建的规则是独立于表之外的，需要将规则绑定到具体的列才能使用。要绑定规则，需要使用 sp_bindrule 的特殊存储过程，其语法如下：

```
sp bindrule [ @rulename = ] 'rule' ,
     [ @objname = ] 'object name'
     [ , [ @futureonly = ] 'futureonly_flag' ]
```

其中，rule 就是创建的规则的规则名，object_name 就是要被绑定规则的对象（列或者用户定义的数据类型）。futureonly_flag 是可选参数，只有在将规则绑定到用户定义数据类型时应用。future_only_flag 的数据类型为 varchar(15)，默认值为 NULL。当此参数设置为 futureonly 时，可以防止具有别名类型的现有列继承新的规则。如果 futureonly_flag 为

NULL，则会将新规则绑定到目前没有规则，或正在使用别名数据类型的现有规则的所有别名数据类型列上。

若被绑定的对象在之前已经绑定了其他规则，则使用 sp_bindrule 将直接替换对象上原有的规则，而不需要先撤销绑定原有的规则。以前面提到的 Sick 表为例，若不使用 CHECK 约束，而是使用规则，则相应的 SQL 脚本如代码 3.28 所示。

代码 3.28　创建并绑定规则

```
CREATE TABLE Sick
(
    SickID int IDENTITY PRIMARY KEY,
    SickName nvarchar(5) NOT NULL,
    BloodType varchar(2) NOT NULL
)
GO
CREATE RULE BloodType --创建规则
AS
@type IN ('A','B','AB','O')
GO
EXEC sp_bindrule 'BloodType','Sick.BloodType' --绑定规则
```

说明：数据列上 CHECK 约束和规则可以同时存在，数据必须同时满足 CHECK 约束和绑定的规则才是有效数据。

在创建规则后可以在 SSMS 对象资源管理器中“可编程性”节点下的“规则”节点看到当前数据库所拥有的规则，如图 3.30 所示。

若需要查看 BloodType 规则绑定到了哪些数据库对象，可以右击 BloodType 规则。在弹出的快捷菜单中选择“查看依赖关系”选项，系统将列出所有绑定了该规则的数据库对象，如图 3.31 所示。

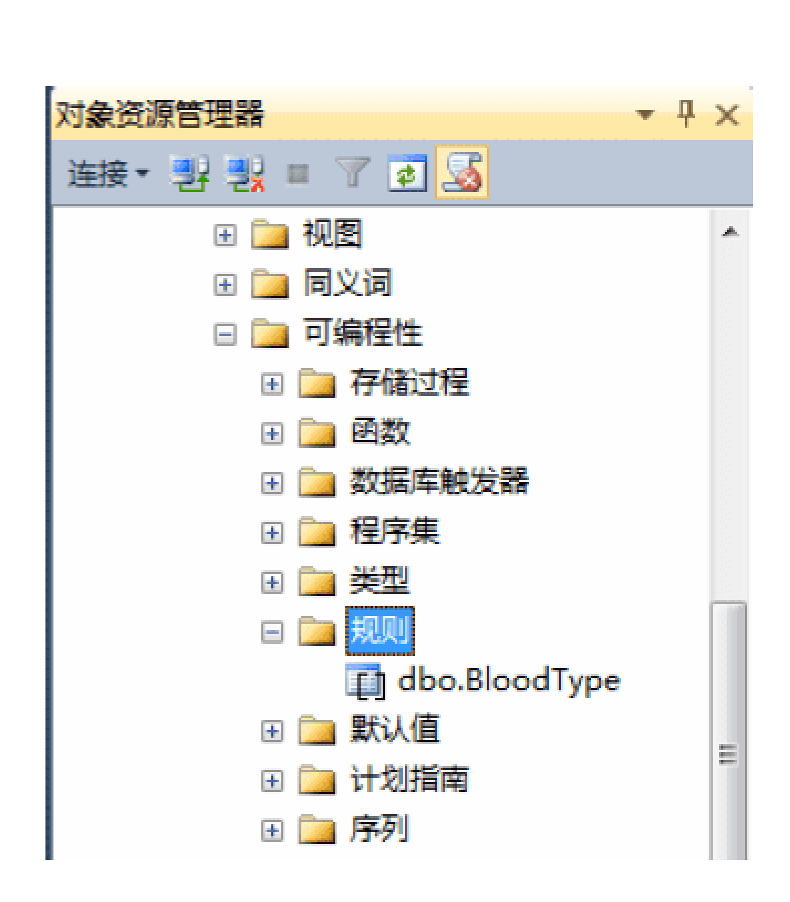


图 3.30　查看规则

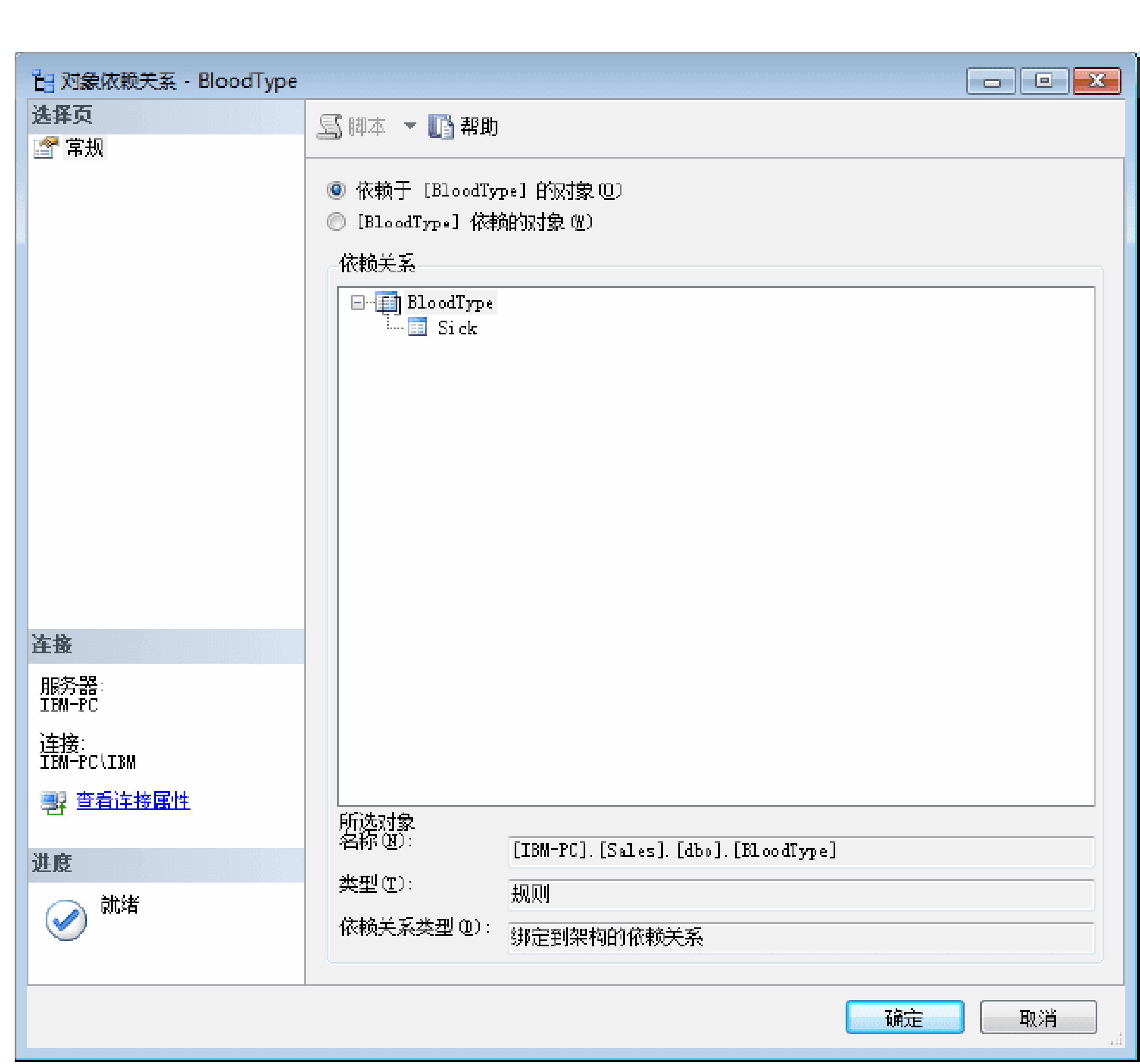


图 3.31　规则的依赖关系

如果想从一列上取消绑定规则，则要使用 sp_unbindrule：

```
EXEC sp_unbindrule 'Sick.BloodType'
```

如果想从数据库中彻底删除规则，则可以使用 DROP 命令：

```
DROP RULE <rule name>
```

注意：在规则绑定到列后，如果没有从列上取消绑定规则，那么该规则是无法删除的。

3.4.9　默认值

这里所说的默认值并不是前面提到的 DEFAULT 约束。默认值和规则一样，是属于数据库早期系统中的概念，现在只是作为向后兼容而保留。默认值和规则一样是独立于表的数据库对象，同样也是通过特定的存储过程绑定到其他对象上。定义默认值的方法和规则的定义方法相似，如下代码所示。

```
CREATE DEFAULT <default name>
AS <default value>
```

在定义好默认值后，接下来就是将该默认值绑定到对象上，使用存储过程 sp_bindefault 绑定默认值。sp_bindefault 的参数与绑定规则的存储过程 sp_bindrule 的参数相同，在此就不再重复介绍。以一个人员表为例，使用默认值将 Age 默认设定为 20，对应的 SQL 脚本如代码 3.29 所示。

代码 3.29　创建并使用默认值

```
CREATE TABLE [Users]
(
    id int IDENTITY PRIMARY KEY,
    UserName nvarchar(10) NOT NULL,
    Age int NOT NULL
)
GO
CREATE DEFAULT Default20 --创建默认值
AS 20
GO
EXEC sp_bindefault 'Default20','Users.Age' --绑定默认值
```

创建默认值后可以通过 SSMS 的对象资源管理器查看当前数据库的默认值。数据库的默认值在“可编程性”节点下的“默认值”节点中，如图 3.32 所示。

若需要知道某个默认值被绑定到哪个数据库对象中，可以右击该默认值，在弹出的快捷菜单中选择“查看依赖关系”选项，系统将弹出“对象依赖关系”对话框，显示被绑定该默认值的数据库对象，如图 3.33 所示。

若要取消绑定的默认值，可以使用 sp_unbindefault：

```
EXEC sp_unbindefault 'Users.Age'
```

如果需要从数据库中删除默认值，可以使用类似的 DROP 命令：

```
DROP DEFAULT <default name>
```

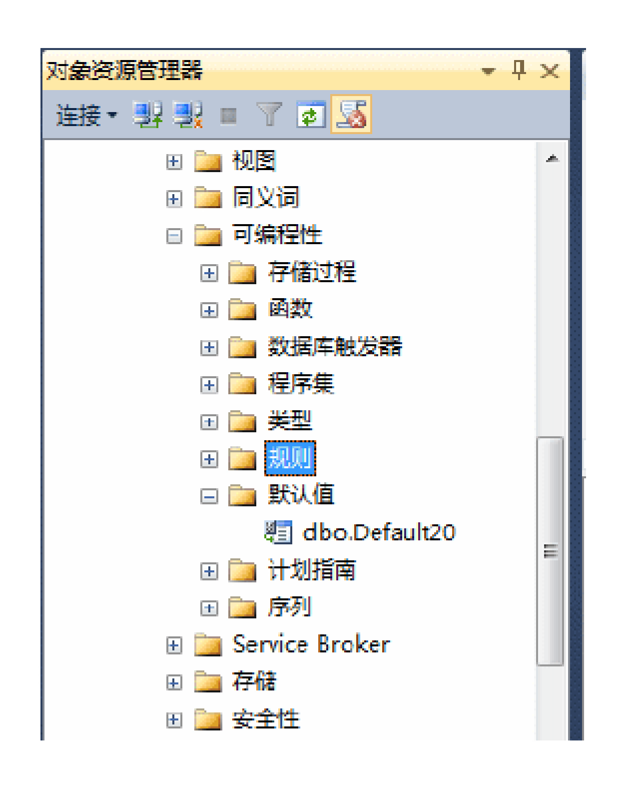


图 3.32　查看默认值

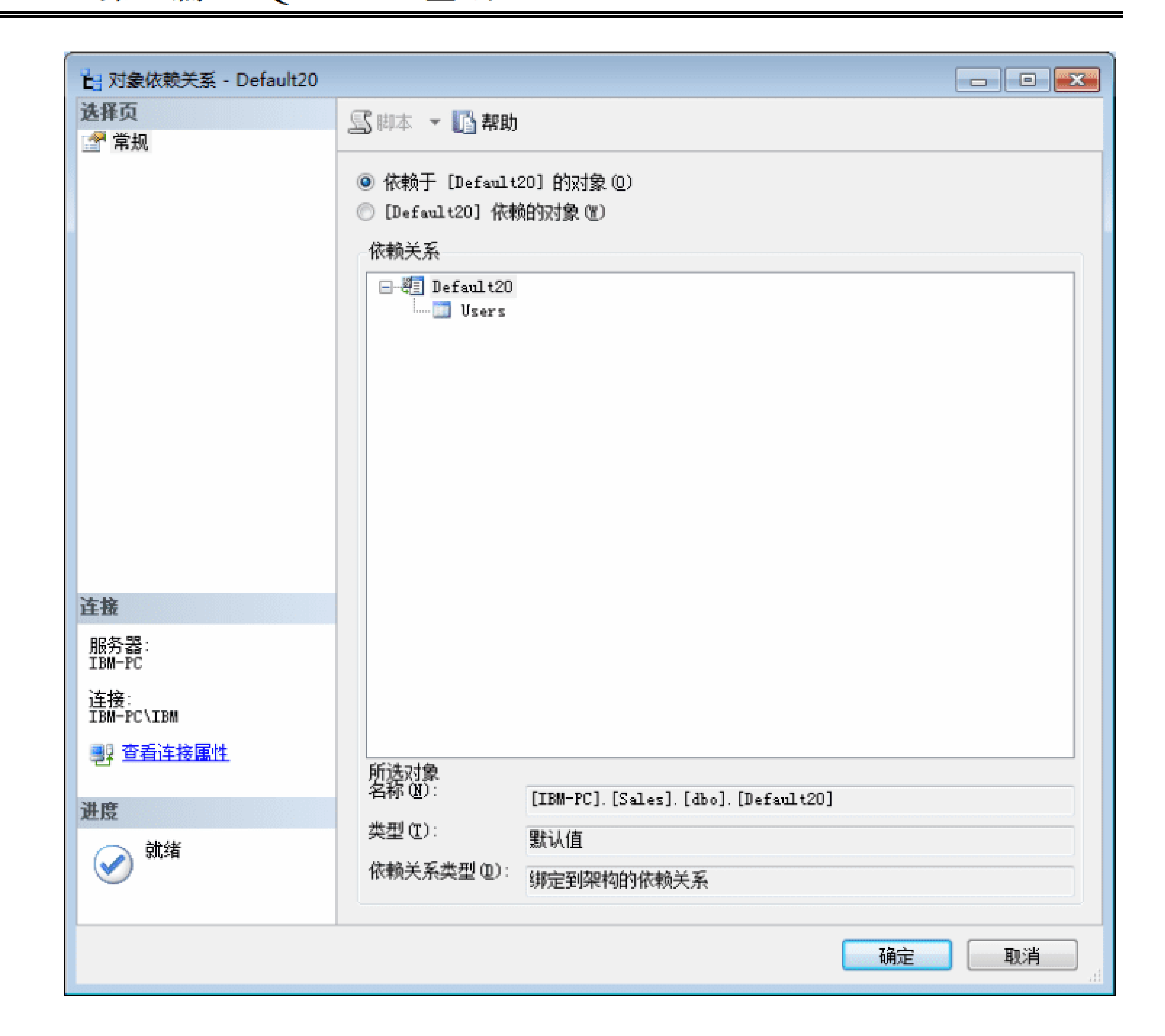


图 3.33　默认值的对象依赖关系

关于在 SSMS 中查看默认值以及默认值的依赖方法与查看规则类似，在此就不做介绍。读者可以自己在 SSMS 中操作一下。

3.4.10　禁用约束

在创建表并同时创建了约束的情况下，以后插入的数据都是符合约束的。但是很多情况是表建立后并记录了大量的数据，之后若希望创建约束，但是数据库中现有记录却有不满足约束的情况，此时约束将会创建失败。由于业务上的原因希望原记录维持原样，此时如果想添加约束，可以在添加约束的 ALTER TABLE 命令后使用 WITH NOCHECK 选项。

例如现在有一个产品管理系统，其中产品表 Product 包含了产品的名称 ProductName 和产品编号 ProductCode。代码 3.30 用于创建该表，并向其中写入两条数据。

代码 3.30　创建 Product 表

```
CREATE TABLE Product
(
    Pid int IDENTITY PRIMARY KEY,
    ProductName nvarchar(50) NOT NULL,
    ProductCode varchar(10) NOT NULL
)
GO --创建表，然后插入数据
INSERT INTO Product(ProductName,ProductCode)
VALUES('产品 A','C01001');
INSERT INTO Product(ProductName,ProductCode)
VALUES('产品 B','A01051');
```

在上面代码中，向 Product 表中写入数据用于表示该表已经使用一段时间，存在一些

历史记录。假设现在业务上需要对新产品进行统一编号，全部使用 SN 开头，而旧数据仍然保持原有编号。此时就需要对 ProductCode 建立约束，而历史数据并不满足约束规则，所以使用 WITH NOCHECK 选项创建约束。具体 SQL 脚本参见代码 3.31。

代码 3.31　使用 WITH NOCHECK 创建约束

```
ALTER TABLE Product
WITH NOCHECK  --不检查历史数据
ADD CONSTRAINT CK_Product CHECK(ProductCode LIKE 'SN%')
GO
INSERT INTO Product(ProductName,ProductCode)
VALUES('产品 C','SS01001');--不满足约束，插入失败
GO
INSERT INTO Product(ProductName,ProductCode)
VALUES('产品 D','SN01051');--满足约束，插入成功
GO
```

除了创建约束时忽略历史数据的情况外，还有一种情况就是约束已经创建成功，但是有一批不符合该约束的数据（从遗留数据库或其他系统中获得的重要数据）必须原封不动地导入到数据库中。完成这种任务的一种方式就是先删除已经建立的约束，将数据导入后再使用 WITH NOCHECK 选项添加约束。实际上 SQL Server 还为用户提供了一种更简单的方式完成此项任务，那就是使用带 NOCHECK 的 ALTER 语句关闭约束。

以前面提到的产品表 Product 为例。在创建数据库时已经建立了 CHECK 约束，之后需要关闭约束，将不符合约束的数据写入数据库中。具体操作如代码 3.32 所示。

代码 3.32　关闭约束

```
CREATE TABLE Product  --创建带有约束的产品表
(
    Pid int IDENTITY PRIMARY KEY,
    ProductName nvarchar(50) NOT NULL,
    ProductCode varchar(10) NOT NULL ,
    CONSTRAINT CK_Product CHECK(ProductCode LIKE 'SN%')
)
GO
INSERT INTO Product(ProductName,ProductCode)
VALUES('产品 A','01001'); --不符合约束的数据无法插入
GO
ALTER TABLE Product
NOCHECK CONSTRAINT CK_Product --禁用约束
GO
INSERT INTO Product(ProductName,ProductCode)
VALUES('产品 A','01001'); --不符合约束的数据允许插入
```

在将历史数据导入完毕后，若需要重新启用约束，只需要将 CHECK 代替 NOCHECK 使用相同的命令即可：

```
ALTER TABLE Product
CHECK CONSTRAINT CK_Product --启用约束
```

3.5　视　　图

视图是一个虚拟的表，其内容由查询定义。同真实表一样，视图包含一系列带有名称的列和行数据。视图在数据库中并不存在于存储中，存在的只是视图的定义，但视图在数据查询中起着重要的作用。本节将主要讲解视图的相关知识。

3.5.1　视图简介

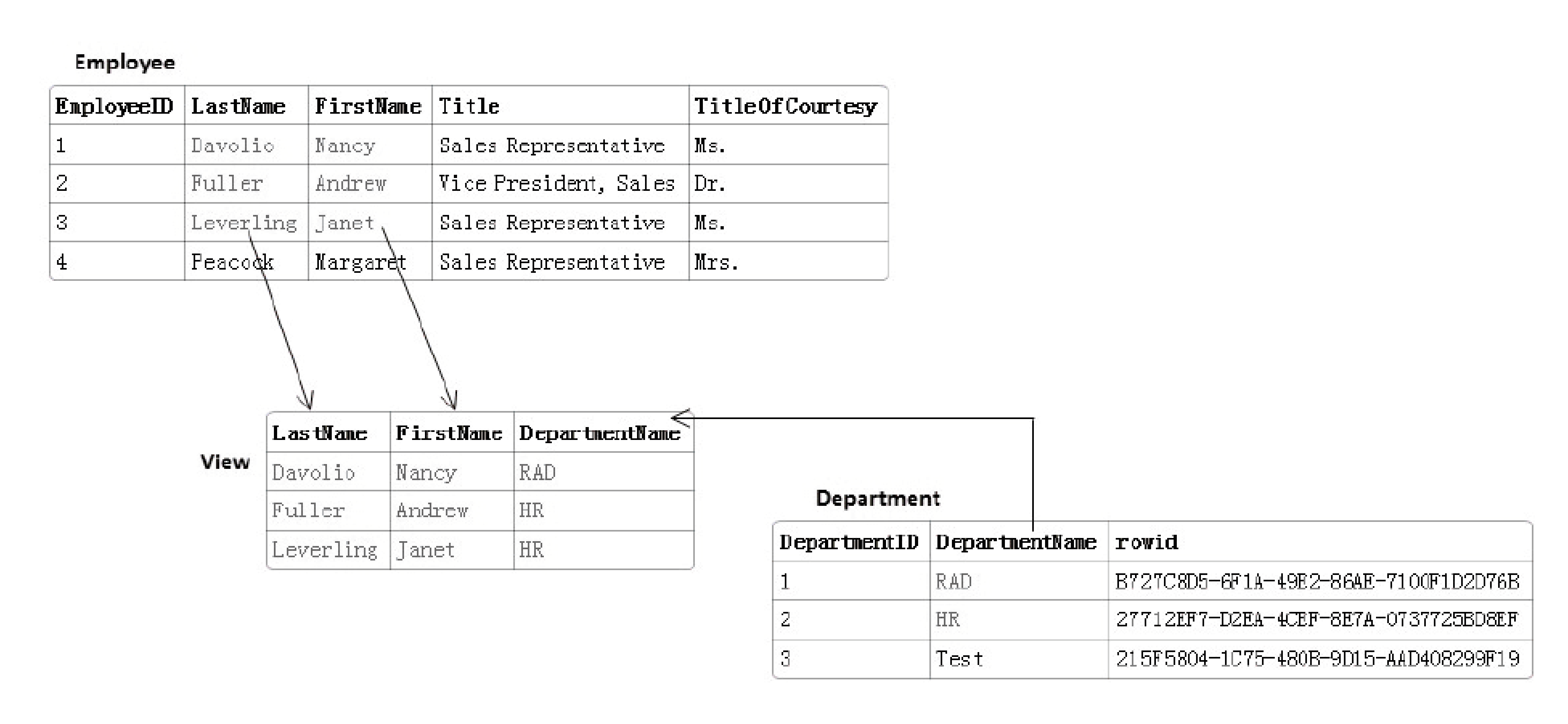
视图由查询定义，可以完成以下工作：

- 降低用户读取数据库数据的复杂性。
- 阻止选择保密列。
- 在数据库中添加索引以改善查询性能。

对于其他持久基表来说，视图的作用类似于筛选。定义视图的筛选可以来自于当前数据库或其他数据库中的一个或多个表，或者其他视图。在后面讲到的跨实例连接可以实现分布式查询，分布式查询也可用于定义使用多个异类源数据的视图。

使用视图的主要优点就是可以定义复杂的查询，并且可以把查询存储在视图定义中。这样，当需要查询的时候就不需要编写复杂的查询脚本，而可以直接调用视图。视图是向用户显示信息的便利方法，视图不会向用户提供超过其需求的信息，也不会向用户提供其不应看到的信息。

图 3.34 显示了在员工表 Employee 和部门表 Department 上建立的视图。用户通过该视图既简化了操作也得到了需要的信息，同时也不会把 rowid 等对用户没有用的信息提供给用户。



Employee

EmployeeID	LastName	FirstName	Title	TitleOfCourtesy
1	Davolio	Nancy	Sales Representative	Ms.
2	Fuller	Andrew	Vice President, Sales	Dr.
3	Leverling	Janet	Sales Representative	Ms.
4	Peacock	Margaret	Sales Representative	Mrs.

View

LastName	FirstName	DepartmentName
Davolio	Nancy	RAD
Fuller	Andrew	HR
Leverling	Janet	HR

Department

DepartmentID	DepartmentName	rowid
1	RAD	B727C8D5-6F1A-49E2-86AE-7100F1D2D76B
2	HR	27712EF7-D2EA-4CEF-8E7A-0737725BD8EF
3	Test	215F5804-1C75-480B-9D15-AAD408299F19

图 3.34　视图

3.5.2　使用 T-SQL 创建视图

创建最简单的视图就是只引用一张表，并且视图从表中取数据时不修改该数据。视图越复杂，视图的定义就越复杂。创建视图的基本语法如代码 3.33 所示。

代码 3.33　创建视图的语法

```
CREATE VIEW <view name> [(<column name list>)]
[WITH ENCRYPTION] [, SCHEMABINDING] [, VIEW_METADATA]
AS <select statement>
WITH CHECK OPTION
```

在上面代码中，第 1 行和第 3 行是必需的，在第 1 行中必须提供视图的名称。此外，在下列情况下必须提供列名。

- 所有列的值都是基于某种计算要插入列的值的操作，而不是直接从基表中复制的值。
- 表的列名相同，在联接查询时出现重名的列。
- 在不需要提供列名的情况下也可以对列进行命名，这样使得列名更易理解。在提供列名时必须对视图中的每个列提供列名，而不能只给部分列提供列名。

上面代码中的第 3 句 AS 关键字不能省略，AS 后的<select statement>便是查询语句。这些查询语句可以执行大量操作，包括从多个表中检索数据、计算数据、限制返回数据类型等。由于查询的重要性和复杂性，将在后面专门进行讲解。此处主要介绍视图，对查询就不做过多介绍。

为了方便读者直观地认识和使用视图，这里将使用 AdventureWorks 作为示例数据库进行操作。代码 3.34 演示了视图的创建和使用方法。该代码创建的视图 wEmployee 包含了 Employee 表中的所有信息，同时还将部门名称 DepartmentName 也展示了出来，而 Department 表中的其他信息则不需要展示给用户。

代码 3.34　创建并查询视图

```
USE AdventureWorks2012
GO
CREATE VIEW vwEmployee --创建视图
AS
SELECT e.*,d.Name AS DepartmentName --联接查询多张表
FROM HumanResources.Department d
INNER JOIN HumanResources.EmployeeDepartmentHistory edh
ON edh.DepartmentID = d.DepartmentID
INNER JOIN HumanResources.Employee e
ON e.BusinessEntityID = edh.BusinessEntityID
GO
```

除了将表作为视图的查询数据源外，还可以将其他视图作为查询数据源。如需要对 Employee 的属性进行扩展，建立 vwEmployeeContract 视图，除了部门名称外还要将职员的个人信息也包括在其中，则可将 vwEmployee 视图作为查询的数据源进行处理，如代码 3.35 所示。

代码 3.35　使用其他视图和表创建视图

```
CREATE VIEW vwEmployeeContract
AS
SELECT e.*,c.FirstName,c.LastName,c.MiddleName
FROM vwEmployee e  --对视图进行联接查询
```

```
INNER JOIN Person.Contact c
ON e.BusinessEntityID = c.BusinessEntityID
GO
SELECT *
FROM vwEmployeeContract --查询新视图
```

说明： SQL Server 在 7.0 版之前视图可联接的表数量最多为 16 个。比如有 3 个视图，每个视图中使用了 6 个联接，那么再将这 3 个视图联接来创建视图将会超出限制而创建失败。但在 7.0 和以后的版本中限制为 256 个表的联接，这样在实际业务系统中就很难超过这个限制了。

3.5.3　使用 SSMS 创建视图

虽然 SSMS 也提供了可视化的界面来创建视图，但是就笔者使用的感受而言，使用 SSMS 创建视图的操作其实并不简单，而且也不灵活，还是用 T-SQL 更方便。虽然如此，作为一种常用的数据库操作，本小节还是讲解一下使用 SSMS 创建视图的操作方法。同样以 AdventureWorks 数据库为例，若要实现代码 3.34 的效果，创建并查询视图的操作如下所述。

（1）在对象资源管理器中展开 AdventureWorks2012 数据库并右击“视图”节点。在弹出的快捷菜单中选择“新建视图”选项，系统将弹出“添加表”对话框，如图 3.35 所示。

（2）由于在要创建的视图中要用到 HumanResources.Department、HumanResources. EmployeeDepartmentHistory 和 HumanResour- ces.Employee 表，所以在“添加表”对话框中，双击 Department（HumanResources）等 3 个表将该表添加到视图中。

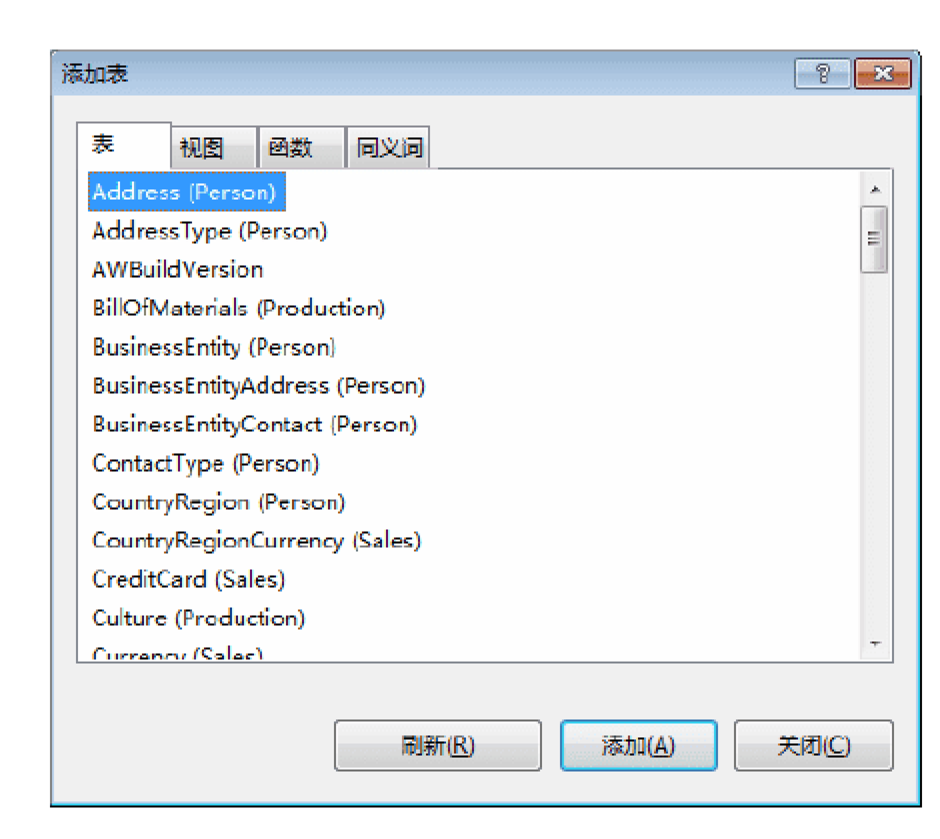


图 3.35　“添加表”对话框

（3）单击“关闭”按钮，系统将进入视图设计窗口，如图 3.36 所示。添加的 3 个表以图形的方式显示在设计窗口上方，而这 3 个表之间的连线是系统通过外键约束自动建立的关联。

（4）在表的图形窗口中列出了表的列名，在每个列名左边有复选框。其中，选中的复选框的列就是要出现在视图中的列。在该设计器中选择需要呈现的列：Employee 表的所有列和 Department 表的 Name 列，如图 3.37 所示。

（5）在视图设计器的中间窗格中显示了要输出的列，在 Name 行的“别名”列中输入要输出的别名 DepartmentName，如图 3.38 所示。

（6）单击工具栏的“保存”按钮，输入该视图的名称，比如 vwEmployee1，视图将保存到数据库中。同时通过对象资源管理器可以查看到该视图，如图 3.39 所示。

（7）在对象资源管理器中右击刚创建的视图 vwEmployee1。在弹出的快捷菜单中选择“打开视图”选项，系统将像呈现表一样，通过表格的方式在 SSMS 的主区域新建一个选项卡来展现该视图。

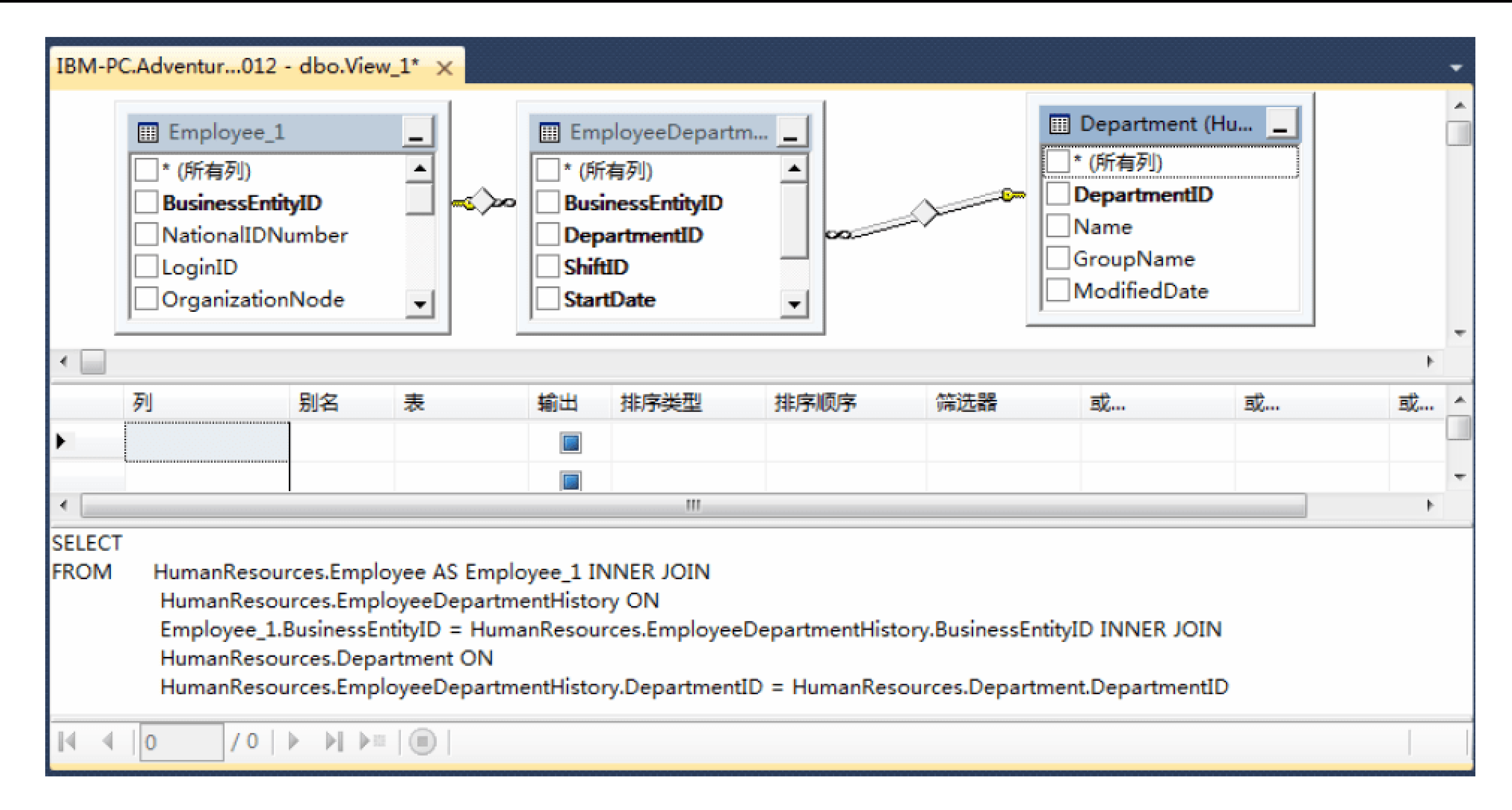


图 3.36　视图设计窗口

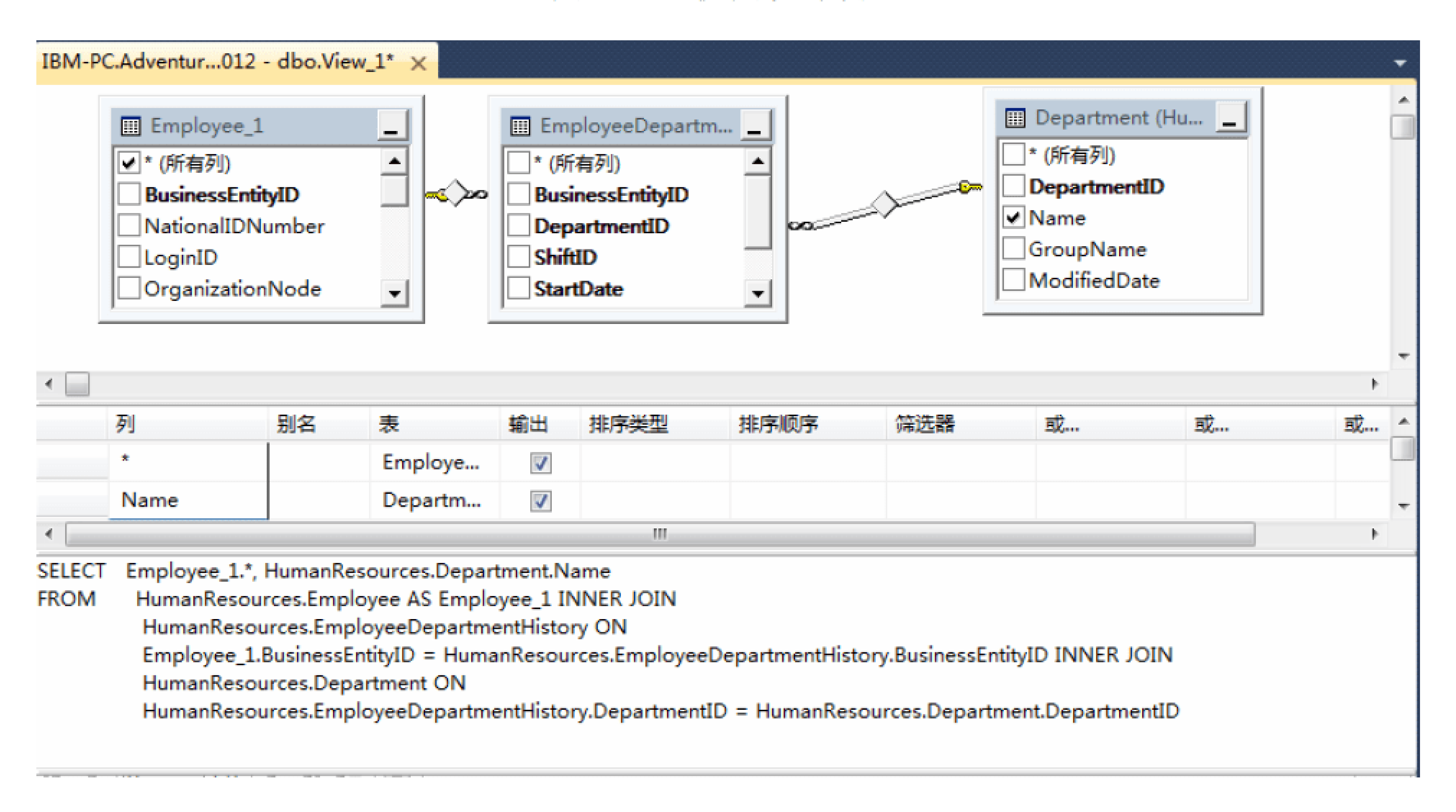


图 3.37　在视图设计器中选择所需的列

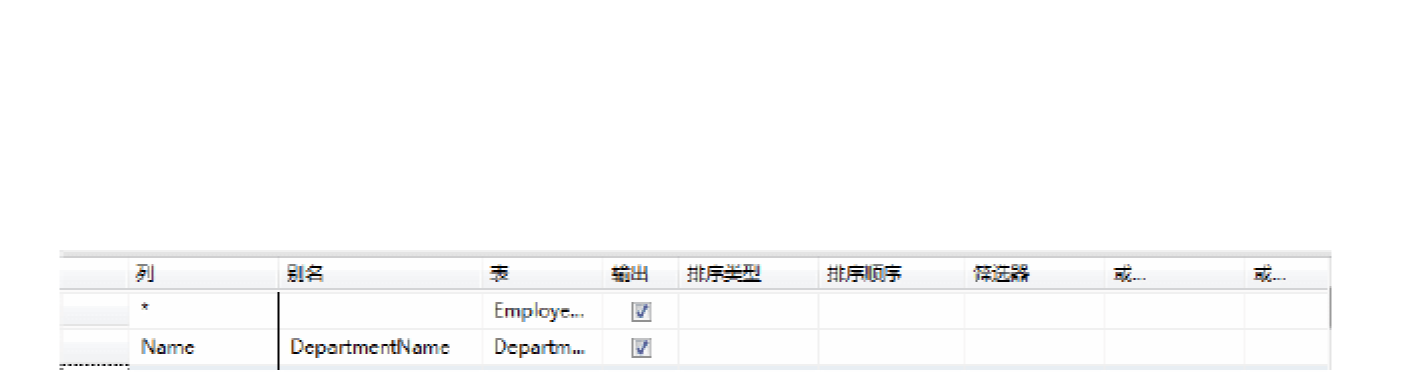


图 3.38　输入列的别名

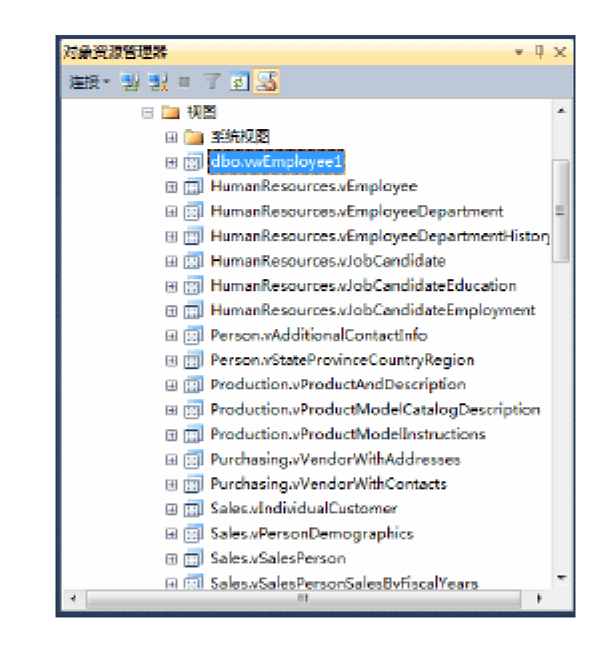


图 3.39　在对象资源管理器中查看视图

技巧： 视图设计器的第 3 个窗格是 T-SQL 语句的编辑区，可以直接在这里编写视图定义中的查询。编写完成后再单击工具栏的“执行 SQL”按钮运行该查询，系统将在视图设计器的第 4 个窗格中显示查询结果。确保查询无误后便可保存该查询。

3.5.4　修改视图

在 T-SQL 中使用 ALTER VIEW 命令修改视图。ALTER VIEW 的语法与创建视图所使用的 CREATE VIEW 语法只是一字之差，其语法如代码 3.36 所示。

代码 3.36　修改视图的语法

```
ALTER VIEW <view name> [(<column name list>)]
[WITH ENCRYPTION] [, SCHEMABINDING] [, VIEW METADATA]
AS <select statement>
WITH CHECK OPTION
```

修改视图时必须指定要修改视图的名称<view name>和修改后视图定义的查询<select statement>。比如要修改代码 3.34 所创建的视图 vwEmployee，在该视图中添加部门分组 GroupName，其 SQL 脚本如代码 3.37 所示。

代码 3.37　修改视图

```
ALTER VIEW [dbo].[vwEmployee]
AS
SELECT e.*,d.Name AS DepartmentName,d.GroupName --增加 GroupName
FROM HumanResources.Department d
INNER JOIN HumanResources.EmployeeDepartmentHistory edh
ON edh.DepartmentID = d.DepartmentID
INNER JOIN HumanResources.Employee e
ON e.BusinessEntityID = edh.BusinessEntityID
```

若使用 SSMS 的可视化操作来修改视图，以前面讲的使用 SSMS 创建的视图 vwEmployee1 为例，为该视图添加部门的 GroupName 列输出的操作如下所述。

（1）在对象资源管理器中右击视图 vwEmployee1，在弹出的快捷菜单中选择“设计”选项，系统将打开 vwEmployee1 的视图设计器。

（2）在视图设计器中找到 Department 表，并选择其列 GroupName 左边的复选框，如图 3.40 所示。

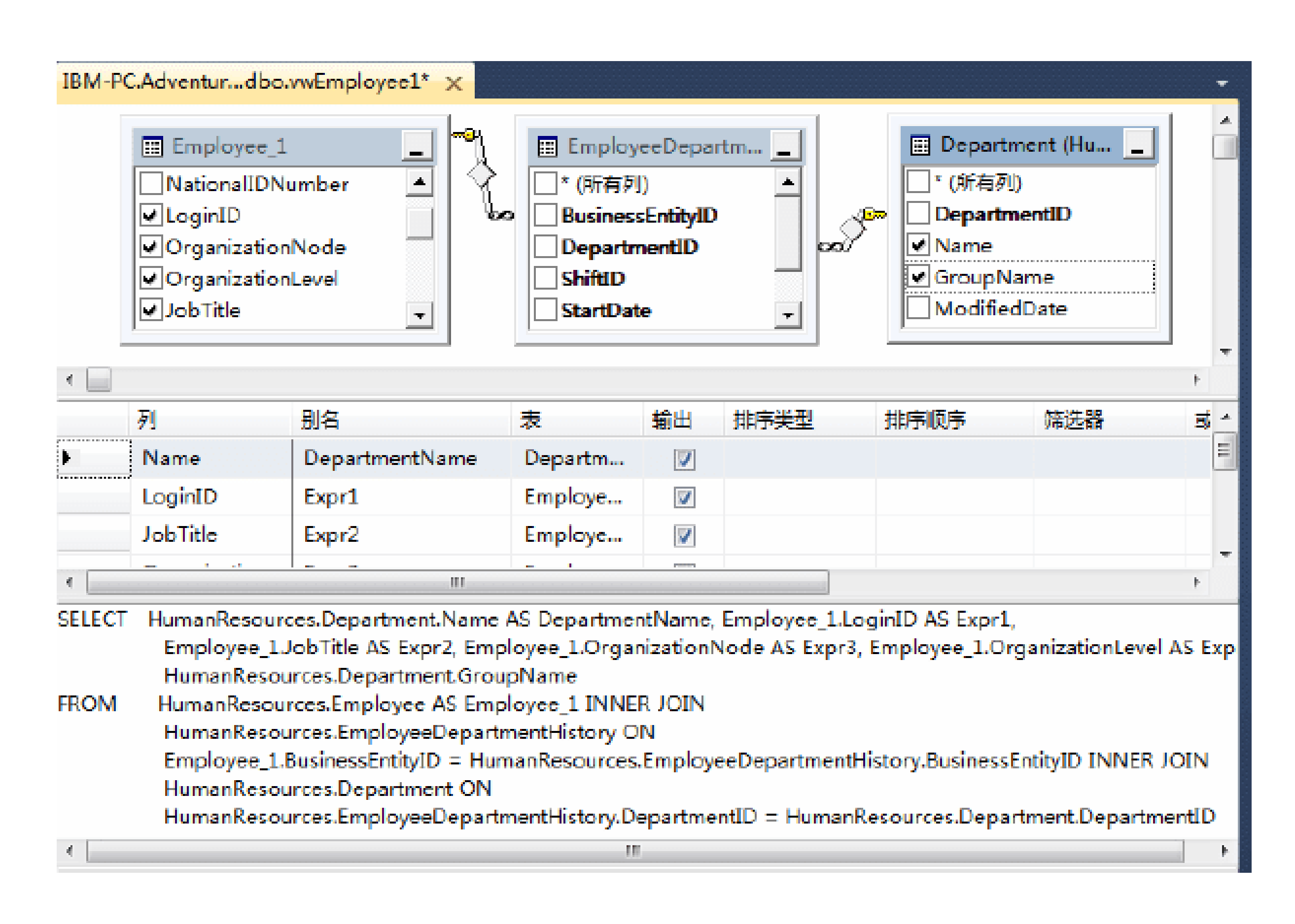


图 3.40　在视图设计器中修改视图

（3）单击工具栏的“保存”按钮或使用快捷键 Ctrl+S 保存对视图的修改。

（4）关闭视图设计器，在对象资源管理器中右击 vwEmployee1，在弹出的快捷菜单中选择“打开视图”选项，可以看到 GroupName 出现在视图中。

注意：由于视图可以被另外的视图作为数据源使用，所以修改视图时要小心。如果删除了某列输出，而该列正好在其他视图中使用，那么在修改该视图后其他关联的视图将无法再使用。

3.5.5　删除视图

在 T-SQL 中使用 DROP VIEW 命令删除视图，其语法格式为：

```
DROP VIEW <view name>,[<view name>,[...n]]
```

如需要将前面创建的视图 vwEmployee 和 vwEmployeeContract 删除，则只需要执行：

```
DROP VIEW vwEmployee, vwEmployeeContract
```

而使用 SSMS 删除视图同样也很简单。在对象资源管理器中选中要删除的视图，使用快捷键 Delete，系统将弹出一个删除确认窗口，单击“确定”按钮即可。

3.6　存储过程

存储过程（Stored Procedure，SP）和函数类似，都是 SQL 面向过程的一种数据库对象。存储过程实际上就是一段独立的 SQL 命令，它存储于数据库中而不是单独的文件中，它有输入参数、输出参数和返回值。本节将主要讲解存储过程和其基本操作。

3.6.1　存储过程简介

存储过程是以特定顺序排列的 T-SQL 语句序列，可为其指定名称，加以编译并保存到 SQL Server 上。一旦由 DBMS 编译并保存，用户可使用应用程序或者其他 SQL 脚本调用并执行存储过程中的语句。其类似于应用程序中的程序调用子程序。存储过程和程序有很多相似之处，具体如下所述。

- 存储过程可以接受输入参数，最终以输出参数或输出结果的格式向调用该存储过程的其他存储过程或 T-SQL 批处理返回值。
- 存储过程可以执行复杂的逻辑操作和运算，可以调用其他存储过程。
- 存储过程不像函数那样将值返回，其向调用过程或批处理返回的是状态值，以指明成功或失败（以及失败的原因）。

注意：存储过程可以调用其他存储过程或自身，但是在 SQL Server 2012 中限制最多可以进行 32 层的调用。

可以使用 T-SQL 的 EXECUTE（简写为 EXEC）语句来运行存储过程。在实际应用开发过程中，推荐在 SQL Server 中使用存储过程而不使用 T-SQL，因为使用存储过程有以下好处。

- 存储过程作为一个数据库对象已在服务器注册。
- 存储过程具有安全特性（例如权限）和所有权链接，以及可以附加到它们的证书。用户可以被授予权限来执行存储过程，而不必直接对存储过程中引用的对象具有权限。
- 存储过程可以加强应用程序的安全性。使用参数化存储过程有助于保护应用程序，降低 SQL 注入攻击的可能性。
- 存储过程允许模块化程序设计。存储过程与调用该存储过程的程序相分离，减少了应用程序与数据库之间的耦合。
- 存储过程在编译后将把执行计划进行存储，在以后的调用中就可以不用像 T-SQL 语句一样每次进行编译然后再执行。由于减少了编译的过程，从而提高了执行的效率。
- 存储过程可以减少网络通信流量。一个需要数百行 T-SQL 代码的操作可以通过将 T-SQL 代码中的逻辑写在存储过程当中，然后由应用程序执行存储过程即可，而不需要在网络中发送数百行代码。
- 存储过程可以进行加密，包含其中的处理逻辑不被其他用户获取。

3.6.2　创建存储过程

创建存储过程与创建其他数据库对象一样使用 CREATE 命令，但是需要使用 AS 关键字。创建存储过程的基本语法如代码 3.38 所示。

代码 3.38　创建存储过程的语法

```
CREATE PROCEDURE|PROC <sp name>
[ { @parameter [ type schema name. ] data type }
       [ VARYING ] [ = default ] [ [ OUT [ PUT ] ] [ ,...n ]
[WITH [RECOMPILE] [,ENCRYPTION]]
[FOR REPLICATION]
AS
<code>
```

从创建存储过程的语法结构来看，该语法仍然具有基本的 CREATE <object type><object name>结构，这是每个 CREATE 语句的主体。与前面讲到的 CREATE 语句不同的是 PROCEDURE 和 PROC 的选择。其实 PROC 就是对 PROCEDURE 的简写，读者可以根据自己的习惯来决定是否使用简写。另外，存储过程的名字也必须符合 SQL 对象的命名规则。

存储过程名后跟的是参数列表。参数列表是可选项，有些存储过程可以不带参数，有些带一到多个参数。参数的声明由参数名、数据类型、默认值和方向 4 个部分构成。当然在声明参数时并不需要都将这 4 个部分写出。一般的参数只有参数名和数据类型两个部分。

- ❑ 参数名必须以@符号开始，参数名的命名规则与其他数据库对象的命名规则类似，只是参数名中不能有空格。
- ❑ 数据类型是在声明变量时必须指出的，并且必须是有效的 SQL Server 数据类型。
- ❑ 默认值是参数与变量存在分歧的地方。通常变量将会被初始化为 NULL，而参数则不是。如果需要定义一个没有提供默认值的参数，那么就需要在调用存储过程时提供一个初始值。
- ❑ 方向是指数据传输的方向，在没有指定的情况下默认为传入。若声明 OUTPUT 或简写 OUT，则表示数据是从存储过程中传出的。

以 AdventureWorks 数据库为例。若要创建一个存储过程，该存储过程返回所有的部门列表，其对应的 SQL 脚本如代码 3.39 所示。

代码 3.39　创建存储过程

```
USE AdventureWorks2012
GO
CREATE PROC GetDepartment  --创建存储过程获得部门列表
AS --以下是存储过程的内容
SELECT *
FROM HumanResources.Department
```

这个存储过程十分简单，而且未带任何参数。如果要执行该存储过程，只需要使用 EXEC 命令：

```
EXEC GetDepartment
```

若现在需要对该存储过程进行扩展，要求根据输入的组名，获得该组下的所有部门。如果没有输入组名，则返回 Sales and Marketing 组下的部门。这里就要使用到参数，同时还为参数指定默认值。具体创建存储过程的 SQL 脚本如代码 3.40 所示。

代码 3.40　创建带参数的存储过程

```
USE AdventureWorks2012
GO
CREATE PROC GetDepartmentByGroupName
@groupName nvarchar(50) ='Sales and Marketing' --定义参数和参数的默认值
AS
SELECT *
FROM HumanResources.Department
WHERE GroupName=@groupName
```

在没有给参数赋值的情况下调用存储过程，存储过程将使用参数的默认值：

```
EXEC GetDepartmentByGroupName
```

执行带参数的存储过程有以下两种格式。

- ❑ 在存储过程名后给出参数的值，参数的值顺序必须与存储过程中定义的参数顺序相同。如输入组名 Research and Development 来执行存储过程：

```
EXEC GetDepartmentByGroupName 'Research and Development'
```

- ❑ 在存储过程名后以“@参数名=参数值”的形式给出参数值。这种方式给出的参数

顺序可以与存储过程中定义的参数顺序不同。具体脚本代码如下：

```
EXEC GetDepartmentByGroupName @groupName='Manufacturing'
```

SSMS 中并未可视化的界面用于设计存储过程，但提供了模板来帮助用户快速地创建存储过程，具体操作如下所述。

（1）在 SSMS 中右击“存储过程”节点。在弹出的快捷菜单中选择“新建存储过程”选项，系统将打开一个创建存储过程的模板。

（2）选择“查询”菜单下的“指定模板参数的值”选项，系统将弹出“指定模板参数的值”对话框，如图 3.41 所示。

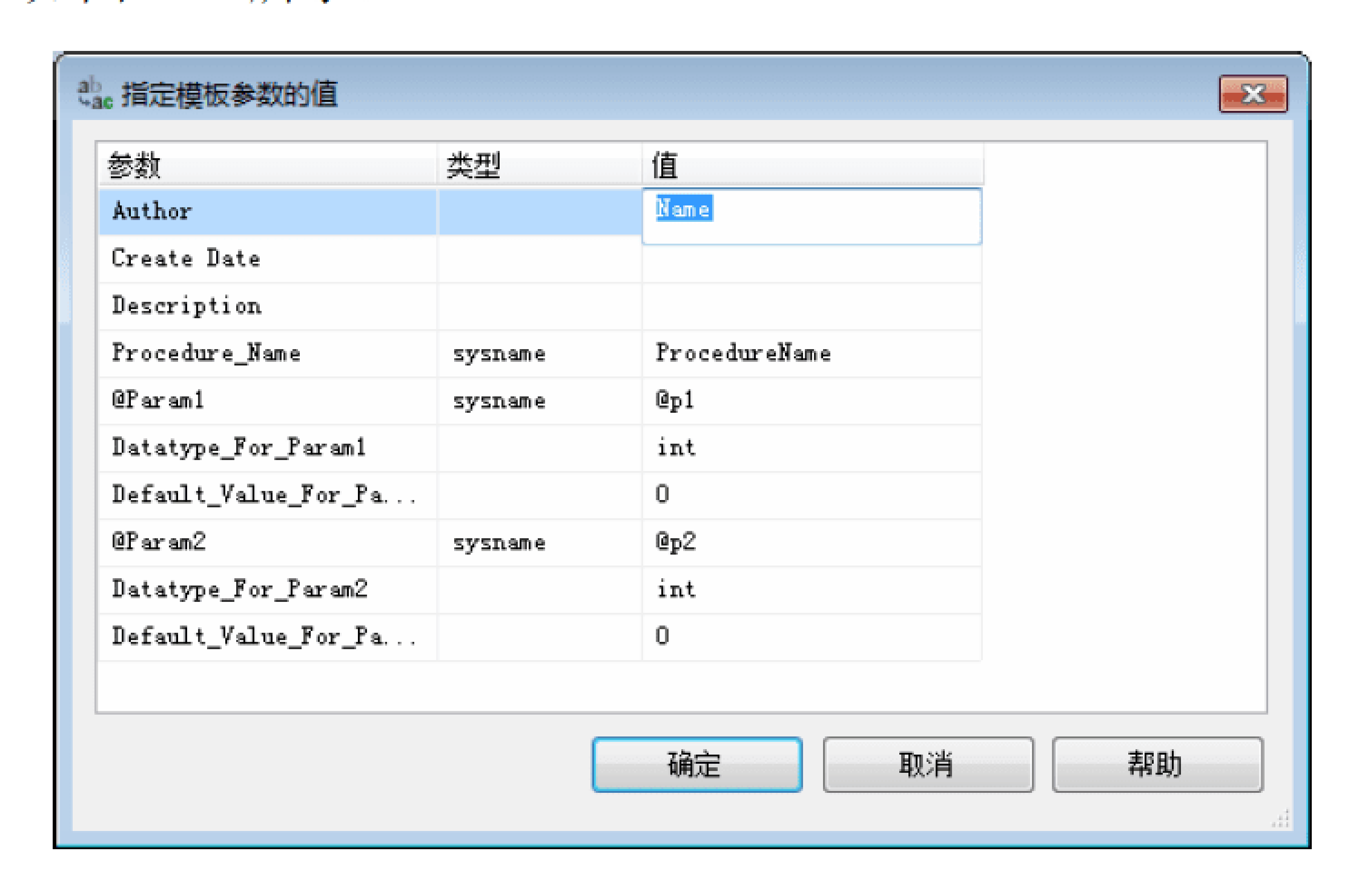


图 3.41　“指定模板参数的值”对话框

（3）在值列中填入需要的数据，单击“确定”按钮，系统将根据模板中的参数值更新 T-SQL 脚本。更新后的 T-SQL 脚本如代码 3.41 所示。

代码 3.41　使用模板生成创建存储过程的 T-SQL

```
--------------------------这里还有一堆注释，此次省略------------------------
SET ANSI NULLS ON
GO
SET QUOTED IDENTIFIER ON
GO
-- =============================================
-- Author:      作者名
-- Create date: 2013-8-1
-- Description: 测试的一个存储过程
-- =============================================
CREATE PROCEDURE GetDepartmentByGroupNameTest
    -- Add the parameters for the stored procedure here
    @groupName nvarchar(50) = ,
      =
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

   -- Insert statements for procedure here
    SELECT @groupName,
```

```
END
GO
```

（4）修改生成的脚本，使其满足实际的需要，完成相应的功能。

（5）单击工具栏的“执行”按钮或者使用快捷键 F5 运行修改后的脚本，完成存储过程的创建。

说明：最终在 SSMS 中，所有存储过程只有通过编写 T-SQL 的方式创建。在实际开发中使用 SSMS 提供的模板来生成存储过程，然后再进行修改比直接使用 T-SQL 创建存储过程的步骤要繁琐。

3.6.3　修改存储过程

如果需要更改存储过程中的语句或参数，可以删除并重新创建该存储过程，也可以通过一个步骤更改该存储过程。删除并重新创建存储过程时，与该存储过程关联的所有权限都将丢失。更改存储过程时，将更改过程或参数定义，但为该存储过程定义的权限将保留，并且不会影响任何相关的存储过程或触发器。修改存储过程使用 ALTER PROC 命令，整个命令的语法格式与创建存储过程的 CREATE PROC 相同，在此不再重复介绍。

以前面创建的存储过程 GetDepartmentByGroupName 为例。现在需要在调用存储过程时必须传入参数，同时返回的结果也要按照部门的 Name 进行排序，则可以修改该存储过程的 SQL 脚本如代码 3.42 所示。

代码 3.42　修改存储过程

```
ALTER PROC [dbo].[GetDepartmentByGroupName]
@groupName nvarchar(50)--去掉默认值，调用该存储过程必须传入参数
AS
SELECT *
FROM HumanResources.Department
WHERE GroupName=@groupName
ORDER BY [Name]  --按部门名排序
```

若使用 SSMS 修改存储过程，只需右击需要修改的存储过程，在弹出的快捷菜单中选择“修改”命令，系统弹出存储过程的定义，然后同样使用 T-SQL 脚本修改。

3.6.4　删除存储过程

删除存储过程与删除视图等数据库对象一样简单，只需执行：

```
DROP PROC|PROCEDURE <sp name>
```

就可删除对应的存储过程。如删除前面创建的存储过程 GetDepartment 的脚本为：

```
DROP PROC GetDepartment
```

使用 SSMS 删除存储过程的操作与删除视图的操作一样。在 SSMS 中选中要删除的存

储过程，使用快捷键 Delete，系统将弹出一个删除确认窗口，单击“确定”按钮即可。

3.6.5　存储过程返回数据

存储过程除了可以被其他存储过程调用外，更多的情况是作为数据库与应用程序的接口，被外部应用程序调用。除了使用 SELECT 命令返回表集外，还可以使用 OUTPUT 参数返回数据，以及存储过程的 RETURN 值的功能。

如果在过程定义中为参数指定 OUTPUT 关键字，则存储过程在退出时可将该参数的当前值返回至调用程序。若要用变量保存参数值以便在调用程序中使用，则调用程序必须在执行存储过程时使用 OUTPUT 关键字。如代码 3.43 创建了一个带 OUTPUT 参数的存储过程，calc 实现两个数相加并将结果传给 OUTPUT 参数。

代码 3.43　使用 OUTPUT 参数的存储过程

```
CREATE PROC calc
@a int,
@b int,
@c int out
AS
SET @c=@a+@b  --输出参数为输入 2 个参数之和
GO
DECLARE @ans int
EXEC calc 1,2,@ans out --执行存储过程
PRINT @ans --将输出参数的值 3 打印出来
GO
DROP PROC calc
```

存储过程可以返回一个整数值（称为“返回代码”），指示过程的执行状态，如代码 3.44 所示，使用 RETURN 语句指定存储过程的返回代码。与 OUTPUT 参数一样，执行存储过程时必须将返回代码保存到变量中，才能在调用程序时使用返回代码值。

代码 3.44　使用 RETURN 返回值的存储过程

```
CREATE PROC compareNumber
@a int,
@b int
AS
IF(@a>=@b)
    RETURN 100
ELSE
    RETURN 200
GO
DECLARE @ans int
EXEC @ans=compareNumber 123,124 --执行存储过程
PRINT @ans --输出 200
GO
DROP PROC compareNumber
```

返回代码通常用在存储过程内的控制流块中，为每种可能的错误情况设置返回代码值。可以在 T-SQL 语句后使用@@ERROR 函数，来检测该语句在执行过程中是否有错误发生。

3.7　用户定义函数

在第 2 章中已经讲到了日期函数、字符串函数和数学函数等系统函数。函数是一个非常重要的数据库工具，在函数的使用中也能够感受到函数带来的方便。本节将主要讲解用户自定义函数及其使用方法。

3.7.1　用户定义函数简介

用户定义函数（User Defined Functions，UDF）同存储过程类似，是一组有序的被预先优化的 T-SQL 语句，并且能够作为一个独立工作单元被调用。UDF 与存储过程的主要区别在于如何返回结果。由于支持不同种类的返回值，所以 UDF 比存储过程的限制要多一些。

使用存储过程可以输入参数也可以得到返回的参数值。前面讲到，存储过程可以返回一个整数值，用于表示成功或失败而不是返回数据。存储过程也可以返回一个结果集，但是如果没有将这些结果集插入到某类表（表变量或临时表）中，则不能使用这些结果集。

但是在 UDF 中只能使用输入参数，而不能使用输出参数。输出参数的概念已经被一个更强的返回值所代替。UDF 中返回的值不像存储过程一样只能是整数数据类型。相反，UDF 不仅可以返回 SQL Server 中的大部分数据类型，也能够返回一张表。返回标量值的函数叫标量值函数，返回表的函数叫做表值函数。

在 SQL Server 中使用用户定义函数有以下优点。

- ❑ 允许模块化程序设计。用户只需创建一次函数并将其存储在数据库中，以后便可以在 T-SQL 语句、存储过程或其他函数中调用任意次。由于用户定义函数以数据库对象的形式存储在数据库中，所以可以独立于程序源代码进行修改，从而降低了程序与数据库之间的耦合。
- ❑ 执行速度更快。与存储过程相似，用户定义函数在编译以后其执行计划便会被缓存，通过缓存计划并在重复执行时重用它可以降低使用 T-SQL 语句的编译开销。
- ❑ 减少网络流量。在定义复杂约束或复杂的 WHERE 条件时，可以使用函数来表示，以减少发送至客户端的数字或行数。

所有用户定义函数都具有相同的由两部分组成的结构：标题和正文。标题和正文之间使用 AS 进行分隔。在 AS 之前的部分为标题，其中定义了：

- ❑ 函数名称，名称之前可以定义函数的架构名称；
- ❑ 输入参数名称和数据类型；
- ❑ 可以用于输入参数的选项；
- ❑ 返回参数数据类型和可选名称；
- ❑ 可以用于返回参数的选项。

AS 之后的部分为正文，其中定义了函数将要执行的操作或逻辑。正文包括以下两者之一：

- ❑ 执行函数逻辑的一个或多个 T-SQL 语句。

❑ 如果是 CLR 函数则定义了.NET 程序集的引用。

3.7.2　创建标量值函数

返回标量值的 UDF 是 SQL Server 在提供 UDF 功能后唯一的 UDF 类型。这种类型的 UDF 更像 SQL Server 的嵌入式函数。这类自定义函数给调用脚本或过程返回一个标量值。

前面已经提到，UDF 最大的优点之一就是不会像存储过程一样限制返回整数值。UDF 能够返回大部分 SQL Server 数据类型，除了 BLOB、游标和时间戳。如果返回的是一个整数值，那么 UDF 和存储过程仍然有所不同，如下所述。

❑ UDF 返回的整数值将可充当一个有意义的数据，而存储过程返回值用来显示执行成功还是失败。

❑ 可以在查询中执行嵌入式 UDF，但存储过程却不能。

创建 UDF 的语法结构如代码 3.45 所示。

代码 3.45　创建 UDF 的语法

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS return data type
    [ WITH <function option> [ ,...n ] ]
    [ AS ]
    BEGIN
                function body
        RETURN scalar expression
    END
```

语法中比较重要的有以下几点。

❑ function_name：用户定义函数的名称。函数名称必须符合有关标识符的规则，并且在数据库，中以及对其架构来说是唯一的。

❑ @parameter_name：用户定义函数中的参数。可声明一个或多个参数。

❑ return_data_type：标量用户定义函数的返回值。对于 T-SQL 函数，可以使用除 timestamp 数据类型之外的所有数据类型（包括 CLR 用户定义类型）。对于 CLR 函数，允许使用除 text、ntext、image 和 timestamp 数据类型之外的所有数据类型（包括 CLR 用户定义类型）。不能将非标量类型 cursor 和 table 指定为 T-SQL 函数或 CLR 函数中的返回数据类型。

❑ function_body：指定一系列定义函数值的 T-SQL 语句。这些语句在一起使用不会产生负面影响（例如修改表）。function_body 仅用于标量函数和多语句表值函数。在标量函数中，function_body 是一系列 T-SQL 语句，这些语句一起使用的计算结果为标量值。在多语句表值函数中，function_body 是一系列 T-SQL 语句。这些语句将填充 TABLE 返回变量。

❑ scalar_expression：指定标量函数返回的标量值。

注意：一个函数最多可以有 1024 个参数。执行函数时，如果未定义参数的默认值，则用户必须提供每个已声明参数的值。

众所周知，人的年龄是随着时间变化的。因此在数据库中一般只保存用户的生日，而不会保存用户的年龄。在需要用户的年龄时可以根据用户的生日和当前的日期计算得出。对此用户可以创建一个标量值函数，输入用户的生日，输出对应的年龄。对应的创建 UDF 的脚本如代码 3.46 所示。

代码 3.46　创建标量值 UDF

```
USE AdventureWorks2012
GO
CREATE FUNCTION dbo.CalcAge(@birthday datetime) --函数名和函数的参数定义
RETURNS int --返回类型
AS
BEGIN
    RETURN year(getdate()) - year(@birthday) --用当前的年份减去出生的年份就是
                                               年龄
END
```

现在这个标量值函数已经创建完成，可以运行一下该函数，测试是否正确：

```
PRINT dbo.CalcAge('1984-2-29')
```

返回 24，说明该函数被正确执行。

注意：与存储过程等其他数据库对象不同，在调用用户定义函数时必须使用 schema_name.function_name 的形式，如果只使用函数名，写成 PRINT CalcAge('1984-2-29')，则系统抛出错误："'CalcAge' 不是可以识别的内置函数名称。"

创建的 UDF 可以嵌入到查询中。如要查询每一个员工的登录名和年龄，并根据年龄从小到大排序，其 SQL 脚本如代码 3.47 所示。

代码 3.47　查询中执行嵌入式 UDF

```
SELECT e.LoginID,dbo.CalcAge(e.BirthDate) AS Age --使用用户定义函数
FROM HumanResources.Employee e
ORDER BY Age
```

SSMS 没有提供创建标量值函数的可视化设计界面，所以标量值函数只有通过 T-SQL 来创建。不过与创建存储过程类似，SSMS 提供了创建标量值函数的模板。模板的使用方法这里就不再介绍了。如果不知道模板的使用读者，可以查看前面创建存储过程的内容。

创建的标量值函数可以在 SSMS 的对象资源管理器中查看。在其中展开具体数据库下的“可编程性”节点，再展开“标量值函数”节点，便可以看到当前数据库的所有用户自定义标量值函数，如图 3.42 所示。

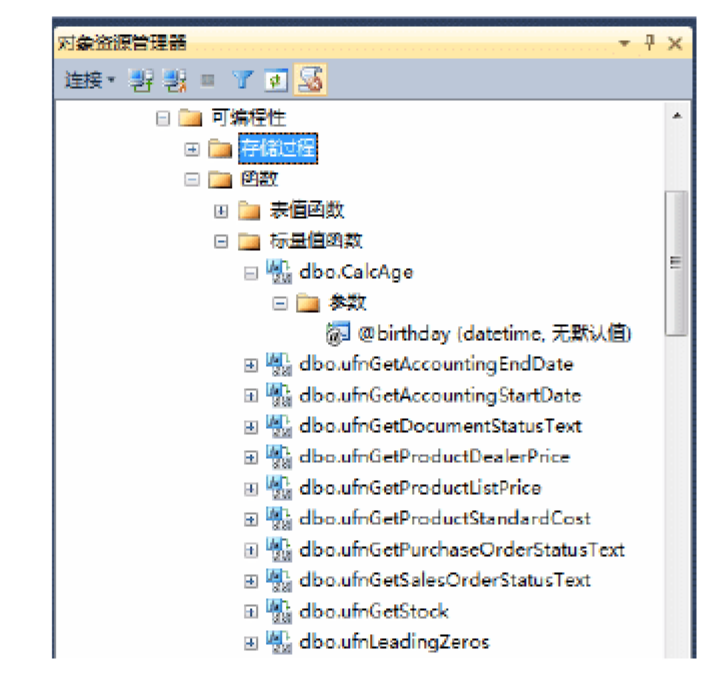


图 3.42　查看标量值函数

3.7.3　创建表值函数

表值函数是 SQL Server 中的新的用户自定义函数，其返回的可以不再是标量值，而是表。在大多数情况下，表值函数返回的表与其他表一样有用，同样可以执行 JOIN，或者跟 WHERE 条件等。创建表值函数的语法与创建标量值函数类似，具体语法如代码 3.48 所示。

代码 3.48　创建表值函数语法

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS TABLE
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    RETURN [ ( ] select_stmt [ ) ]
```

在表值用户定义函数中：

- RETURNS 子句为函数返回的表定义局部返回变量名。RETURNS 子句还定义表的格式。局部返回变量名的作用域位于函数内。
- 函数体中的 T-SQL 语句生成行并将其插入 RETURNS 子句定义的返回变量中。
- 当执行 RETURN 语句时，插入变量的行将作为函数的表格输出返回。RETURN 语句不能有参数。

表值函数中的 T-SQL 语句都无法将结果集直接返回给用户。函数可以返回给用户的唯一信息是该函数返回的表。

以 AdventureWorks2012 数据库为例，若要定义一个表值函数返回一个表，该表包含了员工的登录名和部门，其 SQL 脚本如代码 3.49 所示。

代码 3.49　创建表值函数

```
CREATE FUNCTION udf_GetEmployeeDepartment() --定义函数名和参数
RETURNS TABLE --返回的是一个表
AS
RETURN
(--以下是返回的内容
SELECT e.BusinessEntityID
,e.LoginID,d.DepartmentID,d.Name AS DepartmentName
FROM HumanResources.Employee e
INNER JOIN HumanResources.EmployeeDepartmentHistory edh
ON e. BusinessEntityID =edh. BusinessEntityID
INNER JOIN HumanResources.Department d
ON edh.DepartmentID=d.DepartmentID
)
```

创建好用户定义函数后，若要使用该表值函数，可以将该表值函数当表一样进行查询，唯一不同的是，在使用函数时必须为“架构.函数名”的形式。查询代码 3.49 创建的表值函数如代码 3.50 所示。

代码 3.50　查询表值函数

```
SELECT *
FROM udf GetEmployeeDepartment()
--查询表值函数
SELECT *
FROM udf GetEmployeeDepartment() udf
INNER JOIN HumanResources. EmployeePayHistory a
ON udf. BusinessEntityID =a. BusinessEntityID
--将表值函数与其他表进行 JOIN 操作，这是存储过程无法实现的
```

表值函数与标量值函数以及存储过程都是无法通过 SSMS 的可视化界面进行设计的，只能通过 T-SQL 创建。SSMS 也为创建表值函数提供了 SQL 模板。用户可以使用模板来帮助创建表值函数。

创建的表值函数可以在 SSMS 的对象资源管理器中查看。在对象资源管理器中展开具体数据库下的“可编程性”节点，再展开“表值函数”节点，便可以看到当前数据库的所有用户自定义表值函数，如图 3.43 所示。

图 3.43　查看表值函数

3.7.4　修改用户定义函数

无论是标量值函数还是表值函数，其修改语法与创建语法基本相同。唯一的区别就是，修改用户定义函数使用的是 ALTER 命令，而不是 CREATE 命令。

使用 SSMS 修改用户定义函数比较简单，右击需要修改的函数，在弹出的快捷菜单中选择“修改”选项，系统将根据存储过程的定义自动生成 ALTER FUNCTION 语句。在该语句上进行修改即可。

以前面创建的 CalcAge()标量值函数为例。假设现在由于数据库变动，数据库中不使用 datetime 类型记录用户的生日，而是使用 char(8)来记录年月日，那么就需要对该函数进行修改，使传入的参数为 char(8)，同时函数内容也要做相应变动。修改该标量值函数的 SQL 脚本如代码 3.51 所示。

代码 3.51　修改标量值函数

```
ALTER FUNCTION [dbo].[CalcAge]
(@birthday char(8)) --修改了类型
RETURNS int
AS
BEGIN
    RETURN year(getdate()) - CONVERT(int,left(@birthday,4))
END
```

对于前面创建的表值函数 udf_GetEmployeeDepartment，其返回所有的员工和对应的部门。若需要传入一个参数@groupName，使其返回指定组的员工和部门，则对应的 SQL 脚本如代码 3.52 所示。

代码 3.52　修改表值函数

```
ALTER FUNCTION [dbo].[udf_GetEmployeeDepartment]
(@groupName nvarchar(50)) --增加了参数
RETURNS TABLE
AS
RETURN
(
    SELECT e.BusinessEntityID,e.LoginID,d.DepartmentID,d.Name AS DepartmentName
FROM HumanResources.Employee e
INNER JOIN HumanResources.EmployeeDepartmentHistory edh
ON e.BusinessEntityID=edh.BusinessEntityID
INNER JOIN HumanResources.Department d
ON edh.DepartmentID=d.DepartmentID
WHERE d.GroupName=@groupName
)
```

3.7.5　删除用户定义函数

删除用户定义函数与删除存储过程、视图等数据库对象一样简单，只需执行：

```
DROP FUNCTION { [ schema_name. ] function_name } [ ,...n ]
```

就可删除对应的用户定义函数。如删除前面创建的函数 CalcAge 和 udf_GetEmployee-Department 的脚本为：

```
DROP FUNCTION udf_GetEmployeeDepartment,CalcAge
```

使用 SSMS 删除函数的操作与删除视图的操作一样。在 SSMS 选中要删除的函数，使用快捷键 Delete，系统将弹出一个删除确认窗口，单击“确定”按钮即可。

说明： 如果数据库中存在引用 DROP FUNCTION 的 T-SQL 函数或视图，并且这些函数或视图通过使用 SCHEMABINDING 创建，或者存在引用该函数的计算列、CHECK 约束或 DEFAULT 约束，则 DROP FUNCTION 将失败。如果存在引用此函数并且已生成索引的计算列，则 DROP FUNCTION 将失败。

3.8　触　发　器

Microsoft SQL Server 提供两种主要机制：约束和触发器来强制使用业务规则和数据完整性。前面已经对约束进行了讲解，本节将主要讲解最常用的 DML 触发器的使用，若没有特别说明，本节中介绍的触发器都是指 DML 触发器，其他触发器读者可以自行了解。

3.8.1　触发器简介

触发器（Trigger）为特殊类型的存储过程，可在执行语言事件时自动生效。SQL Server 包括 3 种常规类型的触发器，即 DML 触发器、DDL 触发器和登录触发器。

- ❑ 当服务器或数据库中发生数据定义语言（DDL）事件（比如 CREATE TABLE）时，将调用 DDL 触发器。
- ❑ 当用户登录 SQL Server 实例建立会话时，将触发登录触发器。
- ❑ 当数据库中发生数据操作语言（DML）事件（比如 INSERT）时，将调用 DML 触发器。

DML 事件包括在指定表或视图中修改数据的 INSERT 语句、UPDATE 语句或 DELETE 语句，但不包括 SELECT 语句。因为该语句只是进行查询，并未对数据进行更改。在 DML 触发器中可以操作其他表，还可以包含复杂的 T-SQL 语句。

SQL Server 中将触发器和触发它的语句作为可在触发器内回滚的单个事务对待。如果在触发器中发生了异常，则与该触发器相关的 DML 操作事务即自动回滚。

说明：存储过程可以被直接调用，具有参数和返回值。但是触发器却不行，而且触发器也没有参数和返回值，更不能被直接调用。

DML 触发器在以下几个方面非常有用。

- ❑ DML 触发器可通过数据库中的相关表实现级联更改，实现与外键约束中级联修改同样的功能。不过，通过外键约束的级联修改可以更有效地进行这些更改，所以这种情况下并不推荐使用触发器实现。
- ❑ DML 触发器可以防止恶意或错误的 INSERT、UPDATE 以及 DELETE 操作，并强制执行比 CHECK 约束定义的限制更为复杂的其他限制。在 DML 触发器中可以直接引用其他表中的列进行更复杂的判断；而使用 CHECK 约束则不能直接引用其他表。
- ❑ DML 触发器可以评估数据修改前后的表的状态，并根据该差异采取措施。
- ❑ 一个表中可以定义多个同类 DML 触发器（INSERT、UPDATE 或 DELETE），这样便可在一个数据修改语句中触发多个 DML 触发器采取不同的操作。

根据触发行为的不同，可以将触发器分为以下类型：

- ❑ INSERT 触发器；
- ❑ DELETE 触发器；
- ❑ UPDATE 触发器；
- ❑ 以上几种类型的混合触发器。

注意：有时即使执行的行为看起来像是以上行为的一类，但是也不一定会激活触发器。有日志操作行为的操作才会激活对应的触发器。例如 TRUNCATE TABLE 删除表中的所有数据，但是并没有执行日志操作，所以不会激活 DELETE 触发器。

3.8.2　创建触发器

创建触发器的语法与其他 CREATE 语法相似，但触发器并不能单独存在，而必须在表上创建。创建触发器的语法如代码 3.53 所示。

代码 3.53　创建触发器的语法

```
CREATE TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH ENCRYPTION ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement  [ ; ] [ ,...n ] }
```

该语法中各字段表示的含义为：

- schema_name 和 trigger_name 用于指定触发器的架构和触发器名。触发器名必须遵循 SQL 命名规则而且不能使用“#，##”开头。
- { table | view }用于指定执行 DML 触发器的表或视图，有时称为触发器表或触发器视图。可以根据需要指定表或视图的完全限定名称。视图只能被 INSTEAD OF 触发器引用，不能对局部或全局临时表定义 DML 触发器。
- 其中，AFTER 指定 DML 触发器，仅在触发 SQL 语句中指定的所有操作都已成功执行时才被触发。所有的引用级联操作和约束检查也必须在激发此触发器之前成功完成。如果仅指定 FOR 关键字，则 AFTER 为默认值。不能对视图定义 AFTER 触发器。
- INSTEAD OF 指定执行 DML 触发器而不是触发 SQL 语句。因此，其优先级高于触发语句的操作。不能为 DDL 或登录触发器指定 INSTEAD OF。对于表或视图，每个 INSERT、UPDATE 或 DELETE 语句最多可定义一个 INSTEAD OF 触发器。但是，可以为具有自己的 INSTEAD OF 触发器的多个视图定义视图。
- {[INSERT] [,] [UPDATE] [DELETE] [,]}指定数据修改语句。这些语句可在 DML 触发器对此表或视图进行尝试时激活该触发器。必须至少指定一个选项。在触发器定义中，允许使用上述选项的任意顺序组合。
- sql_statement 指定触发条件和满足触发条件后，将执行 T-SQL 语句中指定的触发器操作。触发器可以包含任意数量和种类的 T-SQL 语句。

DML 触发器使用 deleted 和 inserted 逻辑（概念）表。它们在结构上类似于定义了触发器的表，即对其尝试执行了用户操作的表。在 deleted 和 inserted 表中保存了可能会被用户更改的行的旧值或新值。

说明： 若是 UPDATE 触发，则没有对应的 UPDATE 表。实际上 SQL Server 将视 UPDATE 为先删除后创建，所以声明 FOR UPDATE 的触发器将同时包含了两张特殊的表，即 deleted 表和 inserted 表。

以企业的部门和员工表为例。假设部门表 Department 中包含除了部门的编号 DepartmentID 和部门名 DepartmentName 外，还包含了 EmployeeCount 字段用于表示该部门的员工数，另外还有员工表 Employee。创建 SQL 脚本如代码 3.54 所示。

代码 3.54　创建部门员工表

```
CREATE TABLE Department
(
```

```
    DepartmentID int IDENTITY PRIMARY KEY,
    DepartmentName nvarchar(20) NOT NULL,
    EmployeeCount int NOT NULL DEFAULT(0)
)
GO
INSERT INTO Department(DepartmentName)
VALUES('财务部')
GO
CREATE TABLE Employee
(
    EmployeeID int IDENTITY PRIMARY KEY,
    EmployeeName nvarchar(10) NOT NULL,
    DepartmentID  int  NOT  NULL  FOREIGN  KEY  REFERENCES  Department
    (DepartmentID)
)
```

现在希望在 Employee 上创建一个触发器，使得插入或删除员工的同时能够更新 Department 中的员工数量。由于 UPDATE 操作不会影响员工数，所以这里触发器只需要对 INSERT 和 DELETE 操作激活。创建触发器的 SQL 脚本如代码 3.55 所示。

代码 3.55　创建触发器

```
CREATE TRIGGER UpdateDepartment_EmployeeCount
ON Employee
AFTER INSERT,DELETE
AS
IF(EXISTS(SELECT * FROM inserted))          --执行的是插入操作
BEGIN
    UPDATE Department
    SET EmployeeCount=EmployeeCount+1       --插入操作则对应部门的员工数+1
    WHERE DepartmentID=(SELECT DepartmentID FROM inserted)
END
IF(EXISTS(SELECT * FROM deleted))           --执行的是删除操作
BEGIN
    UPDATE Department
    SET EmployeeCount=EmployeeCount-1       --删除操作则对应部门的员工数-1
    WHERE DepartmentID=(SELECT DepartmentID FROM deleted)
END
```

现在触发器已经创建完成，可以尝试着向 Employee 中插入和删除数据，看 Department 中的员工数是否发生变化。具体操作可以参考代码 3.56 所示。

代码 3.56　验证触发器是否生效

```
SELECT * FROM Department              --初始情况下财务部的员工数为 0
INSERT INTO Employee
VALUES(N'何欢',1)                     --插入 1 名财务部的员工
INSERT INTO  Employee
VALUES(N’晏婉’,1)                     --再插入 1 名财务部的员工
SELECT * FROM Department              --现在财务部的员工数为 2
DELETE FROM Employee                  --删除 1 名财务部的员工
WHERE EmployeeName=N'何欢'
SELECT * FROM Department              --现在财务部的员工数为 1
```

同存储过程和函数的创建一样，SSMS 也只提供了 SQL 模板用于创建触发器。触发器

创建后可以在对象资源管理器中进行查看。如要查看代码 3.55 创建的触发器，可以在对象资源管理器中展开“表”节点下的 Employee 表，然后再展开表下的“触发器”节点，便可以看到 Employee 上创建的触发器，如图 3.44 所示。

图 3.44　查看触发器

3.8.3　修改和删除触发器

修改触发器的语法与创建触发器的语法基本相同。只是修改触发器使用 ALTER TRIGGER 命令，而创建触发器使用的是 CREATE TRIGGER 命令。

对于一个 DML 操作可以定义多个触发器来响应该操作，那么就需要考虑多个触发器的执行顺序问题。可以使用 sp_settriggerorder 指定要对表执行的第一个和最后一个 AFTER 触发器。对一个表只能指定第一个和最后一个 AFTER 触发器。如果在同一个表上定义了 3 个以上的触发器，那么中间其他 AFTER 触发器将随机执行。sp_settriggerorder 的语法如代码 3.57 所示。

代码 3.57　sp_settriggerorder 语法

```
sp_settriggerorder [ @triggername = ] '[ triggerschema. ] triggername'
       , [ @order = ] 'value'
       , [ @stmttype = ] 'statement_type'
       [ , [ @namespace = ] { 'DATABASE' | 'SERVER' | NULL } ]
```

其中，@triggername 参数为要设置顺序的触发器名，第 2 个参数@order 可以为 First、Last 或 None，分别表示是第一个触发器、最后一个触发器和未定义顺序的触发器。第 3 个参数@stmttype 指定激发触发器的 SQL 语句，可以是 INSERT、UPDATE、DELETE、LOGON 或其他 DDL 事件。最后一个参数@namespace 指定所创建的 triggername 是具有数据库作用域还是服务器作用域。例如对于前面创建的触发器 UpdateDepartment_EmployeeCount，设置该触发器为第一个运行，则对应的 SQL 脚本为：

```
sp_settriggerorder @triggername= 'UpdateDepartment_EmployeeCount',
@order='First', @stmttype = 'UPDATE';
```

如果 ALTER TRIGGER 语句更改了第一个或最后一个触发器，将删除所修改触发器上设置的第一个或最后一个属性，并且必须使用 sp_settriggerorder 重置顺序值。

AFTER 触发器就是只有在成功执行触发 SQL 语句之后才能被触发。在执行 DML 语句时必须保证所有与已更新对象或已删除对象相关联的引用级联操作和约束检查。在各级联操作和约束检查都完成后 AFTER 触发器才会被触发。

删除触发器使用 DROP TRIGGER 命令，如需要将前面创建的触发器 UpdateDepartment_EmployeeCount 删除，则只需要执行以下命令：

```
DROP TRIGGER UpdateDepartment_EmployeeCount
```

当然更简单的删除操作方法就是，在 SSMS 中找到需要删除的触发器，然后使用 Delete 键即可。

3.8.4　启用和禁用触发器

默认情况下，创建触发器后会启用触发器。但是在触发器创建后若由于某种原因不希望该触发器运行，则可以使用禁用触发器，禁用触发器不会删除该触发器。该触发器仍然作为对象存在于当前数据库中。但是，当执行编写触发器程序所用的任何 T-SQL 语句时，不会激发触发器。禁用触发器的语法为：

```
DISABLE TRIGGER { [ schema name . ] trigger name [ ,...n ] | ALL }
ON { object_name | DATABASE | ALL SERVER }
```

说明：禁用触发器时如果使用 ALL，则指示禁用在 ON 子句作用域中定义的所有触发器。

以前面用到的 UpdateDepartment_EmployeeCount 触发器为例，现在希望将该触发器禁用（如果已经删除，执行代码 3.55 即可重新创建），那么可以执行以下 SQL 语句：

```
DISABLE TRIGGER UpdateDepartment_EmployeeCount --禁用触发器
ON dbo.Employee
```

由于 Employee 上只有一个触发器，所以也可以将 Employee 上的所有触发器禁用：

```
DISABLE TRIGGER ALL ON dbo.Employee
```

触发器被禁用后，其在 SSMS 中的图标也有所不同。在原来触发器的图标右下角将出现一个向下的小箭头，表示该触发器已经禁用，如图 3.45 所示。

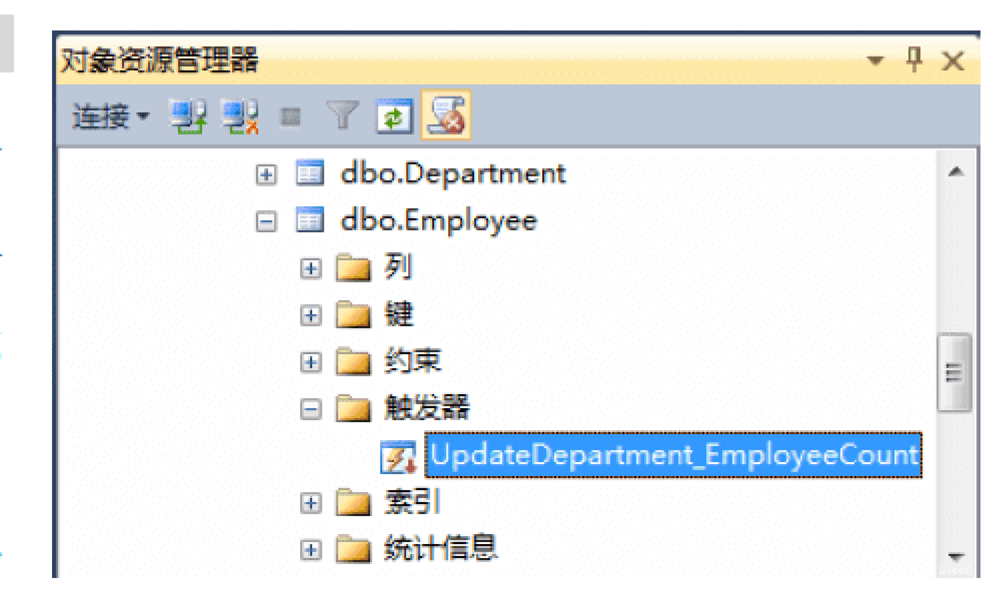


图 3.45　禁用触发器后的触发器表示

在完成了相应的业务后，如希望再将禁用的触发器启用可以使用 ENABLE TRIGGER 命令。其语法为：

```
ENABLE TRIGGER { [ schema name . ] trigger name [ ,...n ] | ALL }
ON { object_name | DATABASE | ALL SERVER }
```

现在需要将前面禁用的 UpdateDepartment_EmployeeCount 触发器启用，其 SQL 脚本为：

```
ENABLE TRIGGER UpdateDepartment_EmployeeCount --启用触发器
ON dbo.Employee
```

或者使用：

```
ENABLE TRIGGER ALL ON dbo.Employee
```

在 SSMS 中禁用或启用触发器将非常简单。在对象资源管理器中右击要禁用或启用的触发器，在弹出的快捷菜单中选择“禁用”或“启用”命令，系统将弹出操作状态窗口，触发器被顺利禁用或启用。

3.9　命名与编码规范

就像写程序时有对于编程语言的编码规范一样，T-SQL 语句的编写，以及数据库对象的命名也有相应的编码规范。但是与编程语言的编码规范不同，T-SQL 语句，以及数据库对象的命名规范并没有得到很好的统一。

3.9.1　命名规范

以下列出一些编码规范以供读者参考。

- 表命名规范：使用项目名或缩写+关注的实体+（关联的实体，视情况而定）。比如：

```
cs_Student blog_Content fa_Department_Employee
```

- 视图命名规范：使用项目名或缩写+vw+关注的实体，或者 vw+项目名+关注的实体。比如：

```
fa_vw_Music vw_forum_Post
```

- 存储过程命名规范：使用项目名或缩写+要执行的操作+被操作的实体+By+传入的参数。比如：

```
fa_getDepartmentByID mf_deleteMusicBySingerID
```

- 索引命名规范：IX/CIX（对应非聚集索引和聚集索引）+索引所在的表的实体名+索引所在字段。比如：

```
IX_Music_MusicName
```

总的来说，数据库对象的命名遵循简单表明类型和意图的原则。也就是说，别人在看到该数据库对象的名字后，不用看具体的内容便可以知道该对象的类型和作用。如表 3.2 列出了几个数据库对象的名字，相信读者也能够一眼看出该对象的类型和作用。

表 3.2　数据库对象命名示例

对　象　名	对 象 类 型	作　　用
eip_Employee	表	存储员工数据
eip_vw_Department	视图	显示部门数据
eip_getEmployeeByDepartmentID	存储过程	通过部门编号获得该部门的所有员工
IX_Department_Name	非聚集索引	在部门表上对部门名建的非聚集索引

注意：有些用户喜欢使用 sp_开头来作为存储过程的命名，这是不对的。因为 SQL Server 中的系统存储过程就是以 sp_开头，用户执行这种命名方式的存储过程时，系统将先到系统存储过程中查找该存储过程，找不到后才会到用户定义的存储过程中查找，从而降低了执行的效率。

3.9.2　SQL 编码规范

SQL 关键字全部使用大写，一般情况下在每个关键字处要换行。注意使用表别名，使用缩进，在逻辑比较复杂的地方应该添加注释，如代码 3.58 所示。

代码 3.58　SQL 编码规范样例

```
DECLARE @id int
SET @id=5
IF(@id>3) --在大于 3 时才进行查询
BEGIN
    SELECT *
    FROM Person.Address a
    INNER JOIN Person.StateProvince sp
    ON sp.StateProvinceID = a.StateProvinceID
    WHERE a.City='Bothell'
END
```

3.10　小　　结

本章主要讲解了常用的 SQL Server 数据库对象的创建、修改和删除的操作。数据库是表、视图、存储过程和函数等数据库对象的容器。一个 SQL Server 实例中包含了多个数据库。表是数据管理的基本单元，所有的业务数据都是以表为容器存放在数据库中。表都是由行和列组成。为了维护业务数据的完整性，可以在表中建立约束。表中的约束包括不为空约束、默认值约束、唯一性约束、主键约束、外键约束和 CHECK 约束。视图是由查询定义的数据库对象。普通视图并不保存数据。视图中的数据通过查询获得。

另外，本章还介绍了存储过程、用户定义函数和触发器等用于开发的数据库对象。最后又简单介绍了数据库命名规范和 SQL 编码规范。

第 4 章　SQL Server 2012 的特色

SQL Server 2012 基于原有 SQL Server 2008 总体架构和用户界面，虽然不像 SQL Server 2005 相对于 SQL Server 2000 的改变那么巨大，但是其新增的特性和功能同样让用户眼前一亮。本章主要概括性地介绍 SQL Server 2012 的一些特性和功能。

4.1　SSMS 增强

无论是进行数据库的管理维护还是数据的更改和查询，SSMS 无疑是使用的最多的一个 SQL Server 管理工具。SQL Server 2012 增加了许多新特性和功能， SSMS 在原有功能的基础上也得到了增强。

4.1.1　键盘快捷方式增强

在对象资源管理器中使用快捷键操作已经成为很多数据库管理人员乐此不疲的事情了。SQL Server 2012 在快捷键方面有了很大的改变，能够让数据库操作人员更加轻松、快速地使用对象资源管理器。

在 SQL Server 2012 中提供了两种快捷键的使用，一种是 SQL Server 2012 的，另一种是 SQL Server 2008 的，默认是 SQL Server 2012 的快捷键。因此，如果数据库操作人员还不熟悉 SQL Server 2012 快捷键的使用，可以选用 SQL Server 2008 的快捷键。更改快捷键的方法是在“工具”菜单下选择“选项”菜单，在其中的环境选项中选择相应的键盘设置即可。

SQL Server 2012 的快捷键的增强主要体现在窗口管理和工具栏的操作、光标移动的快捷键，以及文本选择的操作、代码编辑器中的文本操作等。下面就用表格的形式列出在 SQL Server 2012 新加的快捷键。

1. 窗口管理和工具栏的操作

如表 4.1 所示为 SQL Server 2012 中新增加的窗口管理和工具栏操作部分的快捷键。

表 4.1　窗口管理和工具栏操作新增快捷键

序号	操　作	快　捷　键
1	显示IDE导航器，并选中第一个文档窗口	Ctrl+Tab
2	显示IDE导航器，并选中第一个工具窗口	Alt+F7
3	显示停靠菜单	Alt+减号(-)
4	显示“输出”窗口	Ctrl+Alt+O

续表

序号	操　　作	快　捷　键
5	显示查看历史记录中的上一页。仅在 Web 浏览器窗口中可用	Alt+向左键
6	显示查看历史记录中的下一页。仅在 Web 浏览器窗口中可用	Alt+向右键

2．光标移动的操作

如表 4.2 所示为 SQL Server 2012 中新增加的光标移动操作部分的快捷键。

表 4.2　光标移动操作新增快捷键

序号	操　　作	快　捷　键
1	将光标返回到最后一项	Shift+F8
2	将光标移到文档顶部	Ctrl+PAGE UP
3	将光标移到文档底部	Ctrl+PAGE DOWN

3．文本选择的操作

如表 4.3 所示为 SQL Server 2012 中新增加的文本选择操作部分的快捷键。

表 4.3　文本选择操作新增快捷键

序号	操　　作	快　捷　键
1	将光标左移一个字符，并且扩展选择范围	Shift+向左键
2	将光标左移一个字符，并且扩展列选择范围	Shift+Alt+向左键
3	将光标右移一个字符，并且扩展选择范围	Shift+向右键
4	将光标右移一个字符，并且扩展列选择范围	Shift+Alt+向右键
5	将光标到当前行的开头并扩展列选择范围	Shift+Alt+HOME
6	将光标到行的末尾并扩展列选择范围	Shift+Alt+END
7	返回到导航历史记录中的上一个文档或窗口	Ctrl+减号(-)
8	将光标移至下一个大括号，扩展选择范围	Ctrl+Shift+]
9	前进到导航历史记录中的下一个文档或窗口	Ctrl+Shift+减号(-)
10	交换当前选择范围的定位点和端点	Ctrl+K、Ctrl+A

4．T-SQL键盘调试器快捷键

如表 4.4 所示为 SQL Server 2012 中新增加的 T-SQL 键盘调试器操作部分的快捷键。

表 4.4　T-SQL键盘调试器操作新增快捷键

序号	操作	快捷键
1	单步执行特定语句	Shift+Alt+F11
2	显示下一语句	Alt+NUM
3	启用断点	Ctrl+F9
4	删除断点。仅在“断点”窗口中可用	Alt+F9、D
5	打开“编辑断点标签”对话框。仅在“断点”窗口中可用	Alt+F9、L
6	遇到函数时断开	Ctrl+B
7	显示“并行堆栈”窗口	Ctrl+Shift+D、S
8	显示“并行任务”窗口	Ctrl+Shift+D、K

除了上面列出的 4 类快捷键外，还有文档窗口和浏览器操作、解决方案资源管理器操

作、代码编辑器操作等快捷键。需要详细了解的读者可以参考 http://msdn.microsoft.com/zh-cn/library/ms174205.aspx 中的文档。

4.1.2　查询编辑器增强

查询编辑器是每一个数据库管理员和操作者都经常使用的工具之一。它的便捷和智能得到了多数用户的认可，在 SQL Server 2012 中查询编辑器部分又得到了进一步的改进。在 SQL Server 2012 中查询编辑器主要是在调试功能部分的断点设置上有所增强。具体体现在如下两个方面。

1．指定断点条件

所谓指定断点条件，是指根据 T-SQL 表达式的计算结果来设置断点。但是，T-SQL 表达式的结果必须是布尔值。如果用于指定断点条件的语法无效，则会立即出现一条警告消息。如果用于指定条件的语法有效但语义无效，则会在第一次命中断点时显示一条警告消息。在这两种情况下，当命中无效断点时，调试器都将中断执行。指定条件的设置方法分为如下两个步骤。

（1）打开条件对话框。在查询编辑器窗口中，任意编写一段代码，然后在其代码前面设置断点。右击设置的断点，弹出如图 4.1 所示的右键菜单。选择其中的“条件”选项即可打开条件对话框，如图 4.2 所示。

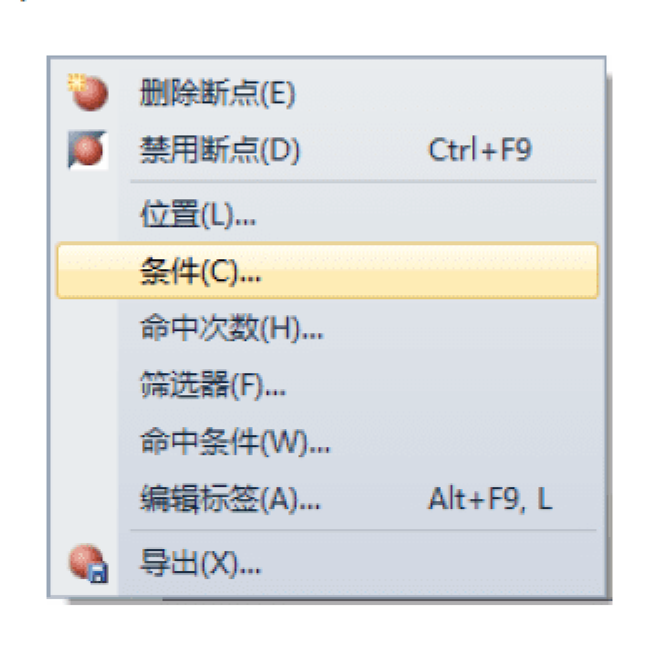


图 4.1　选择断点的右键菜单

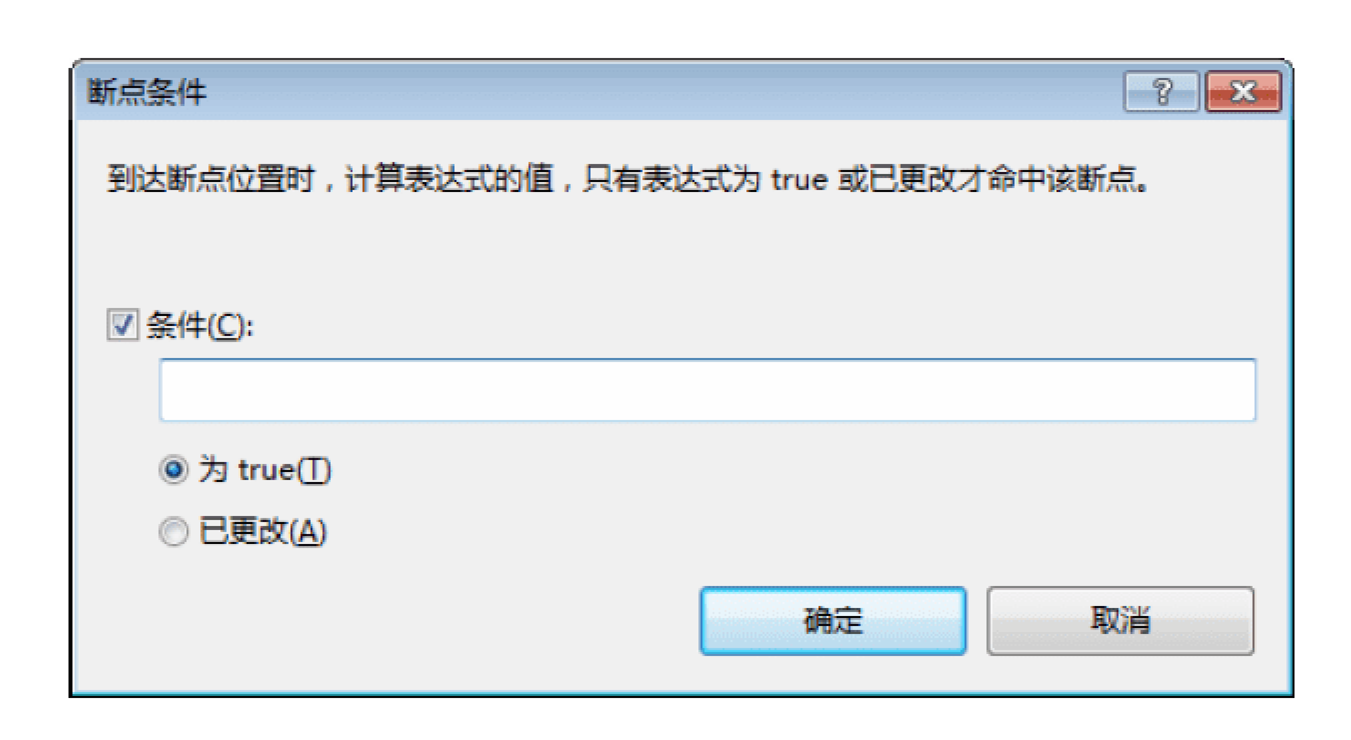


图 4.2　设置条件对话框

（2）设置条件。在图 4.2 所示的界面中的文本框中，任意输入条件表达式，并单击“确定”按钮，即可完成断点条件的设置。断点条件设置完成后，当代码运行到断点处，如果

满足设置的条件，那么，代码运行就被中断了。

2．指定命中计数

所谓命中计数，就是指当代码运行时访问到断点处的次数。在设置命中次数时，可以选择总是中断、当命中计数等于指定值时中断、当命中计数等于指定值的倍数时中断、当命中计数大于或等于指定值时中断。

除了上面详细讲解的两个断点操作之外，还增加了指定断点筛选器、编辑断点位置等新功能。此外，在“监视”窗口和“快速监视”中也支持监视 Transact-SQL 表达式了。

4.2　新增数据类型和视图

SQL Server 2012 中主要是对空间数据类型有所增强，包括圆弧类型、圆球类型的扩展，以及其相关方法的扩展。

4.2.1　圆弧类型的增强

SQL Server 2008 中就提供了圆弧类型，在 SQL Server 2012 里又对圆弧类型进行了改进。在 SQL Server 2012 中增加了如下几个圆弧类型的子类型。

- CircularString，是零个或多个连续圆弧线段的集合。
- CompoundCurve，是几何图形或地域类型的零个或多个连续 CircularString 或 LineString 实例的集合。
- CurvePolygon，是由一个外部边界环，以及零个或多个内环界定的在拓扑结构上闭合的图面。

下面就分别讲解这几个数据类型的具体使用方法。

1．CircularString类型

每一种数据类型都有它的应用范围，那么，CircularString 所代表的类型必须具有如下几个特征。此外，CircularString 类型也可以为空。

（1）必须至少包括一个圆弧线段，也就是至少包括三个点，并且必须是奇数个点。如果有直线段，那么这些直线段必须由三个共线点定义。

（2）序列中的每条圆弧线段的最后一个端点必须是序列中后一条线段的第一个端点。

（3）不能有一条线段与自身线段重合。

使用 CircularString 类型创建一个具有一条圆弧线段的 CircularString 实例，如代码 4.1 所示。

代码 4.1　创建一个圆弧线段的实例

```
DECLARE @g geometry;
SET @g = geometry:: STGeomFromText('CIRCULARSTRING(2 0, 1 1, 0 0)', 0);
SELECT @g.ToString();
```

2．CompoundCurve类型

CompoundCurve 类型能够应用的实例必须都是接受的圆弧线段实例，并且实例中的所有圆弧线段都连接在一起。也就是说，每个后续圆弧线段上的第一个点与前一个圆弧线段上的最后一个点相同。

使用 CompoundCurve 类型存储一个半圆的代码如 4.2 所示。

代码 4.2　使用 CompoundCurve 类型存储半圆

```
DECLARE @g geometry;
  SET @g = geometry::Parse('COMPOUNDCURVE(CIRCULARSTRING(0 2, 2 0, 4 2), (4
2, 0 2))');
SELECT @g.STLength();
```

3．CurvePolygon类型

CurvePolygon 类型所接受的圆弧环要满足如下 3 个条件。

（1）是接受的 LineString、CircularString 或 CompoundCurve 实例。

（2）至少由四个点组成。

（3）起点和终点具有相同的 X 和 Y 坐标。

使用 CurvePolygon 类型实例化一个几何图形，代码如 4.3 所示。

代码 4.3　使用 CurvePolygon 类型实例化几何图形

```
DECLARE @g geometry = 'CURVEPOLYGON(CIRCULARSTRING(2 4, 4 2, 6 4, 4 6, 2
4))'
```

除了新添加的数据类型之外，在 SQL Server 2012 中还添加了一些新的方法操作圆弧。比如，创建一个缓冲对象的 BufferWithCurves()，获取迭代圆弧边的列表的 STNumCurves() 和 STCurveN()方法，用于通过默认和用户指定容差内的线段来模拟圆弧的 STCurveToLine() 和 CurveToLineWithTolerance()方法。

4.2.2　geography 类型的增强功能

在 SQL Server 2008 中，地理功能仅限于比逻辑半球略小。在 SQL Server 2012 中，它们可以与整个地球一样大。可以构造一种新的对象类型（称为 FULLGLOBE），也可以作为操作结果收到这种对象。那么，针对地理功能的扩展，相应地也增加了如下操作方法。

（1）STIsValid()和 MakeValid()：由于在 SQL Server 2012 中允许在地理类型中插入无效对象，可以使用这两个方法采用与几何图形类型相似的方式检测和更正无效的地理对象。

（2）ReorientObject()：重新确定多边形的方向。

（3）STWithin()、STContains()、STOverlaps()和 STConvexHull()方法，以前仅适用于几何图形类型，但现在对于地理类型添加了这些方法。

4.2.3　新添加或修改的视图

系统视图有利于数据库管理员更好地管理数据库，同时也可以为编程人员提供便利的

开发方法。在 SQL Server 2012 中主要添加和修改了如下 6 个系统视图。

- sys.dm_exec_query_stats：用于返回 SQL Server 2012 中缓存查询计划的聚合性能统计信息。在 SQL Server 2012 中在原有视图的基础上添加 4 列，分别是 total_rows、min_rows、max_rows 和 last_rows，可以用于分隔那些从出现问题的查询中返回大量行的查询。需要注意的是，使用该视图需要对服务器具有 VIEW SERVER STATE 权限。
- sys.dm_os_volume_stats：用于返回有关 SQL Server 2012 中存储指定数据库和文件的操作系统卷（目录）的信息。通过使用该视图，可以检查物理磁盘驱动器的属性，或返回有关目录的可用空间的信息。该视图同样需要对服务器具有 VIEW SERVER STATE 权限。
- sys.dm_os_windows_info：用于返回一个显示 Windows 操作系统版本信息的行。该视图同样需要对服务器具有 VIEW SERVER STATE 权限。
- sys.dm_server_memory_dumps：用于返回内存转储文件的路径和名称、创建时间等信息。转储类型可以是小型转储、所有线程转储或完整转储，其扩展名是.mdmp。
- sys.dm_server_services：用于返回 SQL Server、全文和 SQL Server 代理服务的名称、服务启动的模式等信息。该视图同样需要对服务器具有 VIEW SERVER STATE 权限。
- sys.dm_server_registry：用于返回 Windows 注册表中的配置和安装信息。该视图同样需要对服务器具有 VIEW SERVER STATE 权限。

4.3　新的开发特性

SQL Server 2012 除了对空间数据类型有所增强之外，还在开发上提供了更多的新特性，包括列存储索引、文件表、T-SQL 语法、函数和视图等。对于 T-SQL 语法以及函数部分的新特性将在第 13 章中详细讲解。

4.3.1　列存储索引

列存储索引可以大幅提高查询检索的速度，主要应用在数据仓库的查询和统计中。使用列存储数据具有如下 3 个主要特点。

（1）分列数据格式。每次对一个列的数据进行分组和存储。这样，SQL Server 查询处理可以利用新的数据布局，并可以显著改进查询执行时间。

（2）加快查询速度。使用列存储后，每次只查询需要的列，并且列中的数据都是经过压缩处理的，因此，大大加快了查询速度。

（3）表无法更新。在 SQL Server 中，是无法对列存储索引形式的数据表进行更新的。

创建列索引的简单语法形式如下所示。

```
CREATE [NONCLUSTERED] COLUMNSTORE INDEX index name
ON <object> ( column[1,...n ] )
```

其中的参数说明如下所述。

- [NONCLUSTERED]：创建非聚集索引时使用该选项。
- index_name：索引的名字。
- <object>：是数据表的名字。
- column [1,...n]：数据表中的列名，最多可以使用 1024 个列。

但是，创建列索引的列数据类型也是有要求的，必须为以下数据类型才可以。如表 4.5 所示。

表 4.5　创建列索引时列的数据类型

序号	数 据 类 型	描　　述
1	char、varchar	字符串类型，1个字符占1个字节
2	nchar、nvarchar	字符串类型，1个字符占2个字节，但是varchar(max)和nvarchar(max)类型除外
3	decimal、numeric、float、real	小数类型，但是精度大于18位的除外
4	int、bigint、smallint、tinyint、bit	整数类型
5	money、smallmoney	货币类型

下面就使用创建列索引的语法，为订购信息表（orderinfo）创建一个列存储索引。订购信息表的表结构如表 4.6 所示。

表 4.6　订购信息表

序号	字　段　名	类　　型	描　　述
1	orderid	int	订购编号
2	productid	int	产品编号
3	procount	int	订购数量
4	userid	int	订购人编号

为订购信息表中的产品编号列（productid）创建列索引的代码，如代码 4.4 所示。

代码 4.4　创建列索引

```
CREATE NONCLUSTERED COLUMNSTORE INDEX csindex orderinfo
ON orderinfo(productid);
```

说明：通过上面的代码可以看出创建列索引也是很容易的，但是列索引并不适用于所有的数据表中所有的列，它对列有如下 3 个重要限制。

（1）只有非聚集列存储索引才能用。

（2）不能是唯一索引，不能包含主键或外键，也不能包含稀疏列。

（3）不能基于视图或索引视图创建。

4.3.2　文件表

文件表（FileTable）功能建立在 SQL Server FILESTREAM 技术的基础上，为 SQL Server 中存储的文件数据提供 Windows 文件命名空间，以及与 Windows 应用程序的兼容性支持。

文件表可以称为是一种专用的用户表，主要用于存储 FILESTREAM 数据的预定义架构，以及文件和目录层次结构信息、文件属性信息。它具有如下两个主要功能。

（1）表示目录和文件的一种层次结构。该层次结构以创建 FileTable 时指定的根目录为起点。在 FileTable 中每一行代表一个文件或目录。

（2）强制执行某些系统定义的约束和触发器以维护文件命名空间语义。

下面介绍文件表的创建、修改，以及删除的基本操作。

1. 创建文件表

创建文件表既可以使用语句创建，也可以使用企业管理器来创建。下面就分别使用这两种方法来创建文件表。

1）使用语句来创建文件表

创建文件表与普通数据表的语法略有不同，它是不带任何字段的数据表。创建的语法如下所示。

```
CREATE TABLE table name as FileTable
WITH
(
  FILETABLE_DIRECTORY='',
  FILETABLE_COLLATE_FILENAME=''
)
```

其中的参数说明如下所述。

- table_name：表名。
- FILETABLE_DIRECTORY 项：指定充当存储在 FileTable 中的所有文件和目录的根目录的目录。
- FILETABLE_COLLATE_FILENAME 项：指定要应用于 FileTable 的“名称”列的排序规则名称。

使用上面的语法创建文件表，如代码 4.5 所示。

代码 4.5　创建文件表

```
CREATE TABLE FileTable_test AS FileTable
   WITH (
        FileTable Directory = 'testtable',
        FileTable_Collate_Filename = database_default
       );
```

通过上面的语句就可以在相应的数据库中创建文件表了。

创建文件表时经常会出现一些问题，下面就列举了两个常见的问题，并简单说明处理方法。

（1）如果读者创建文件表的数据库中没有启用 FILESTREAM 选项，就会出现图 4.3 所示的错误。

从图 4.3 中就可以看出是由于在数据库 TestDB1 中没有启用“FILESTREAM 文件组”。在数据库中启用 FILESTREAM 通常有两种方法，一种是直接在创建数据库时加上 FILESTREAM 文件组，另一种是在修改数据库时添加 FILESTREAM 文件组。另外，也可以借助 SSMS 工具来创建。

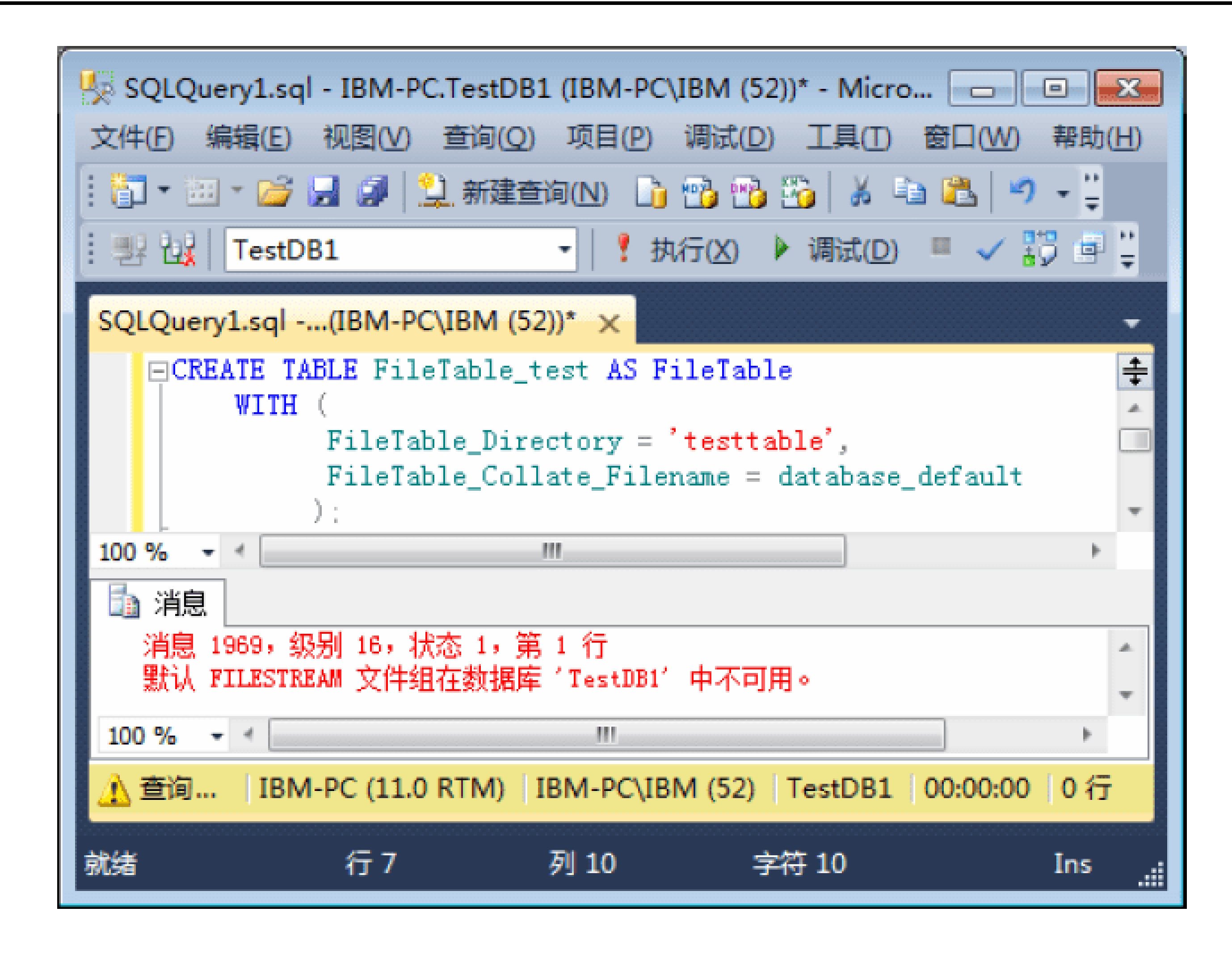


图 4.3　在没启用 FILESTREAM 文件组的数据库上创建文件表

因此，如果要将文件表创建在数据库 TestDb 中，如代码 4.6 所示。

代码 4.6　创建数据库 TestDb

```
CREATE DATABASE TestDb
  ON
  PRIMARY ( NAME = TDb1,
      FILENAME = 'd:\data\TDb1.mdf'),
  FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM(NAME=TDb3,
      FILENAME = 'd:\data\TDb3')
  LOG ON  ( NAME = TDblog1,
      FILENAME = 'd:\data\TDblog1.ldf')
```

如果要在修改数据库时，启用 FILESTREAM 文件组也是可以直接用代码操作的。在 TestDb1 数据库中添加一个 filestream 文件组，如代码 4.7 所示。

代码 4.7　为数据库 TestDb1 添加 FILESTREAM 文件组

```
ALTER DATABASE TestDb1
ADD FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM
```

（2）创建启用 FILESTREAM 文件组时出现如图 4.4 所示的错误提示。

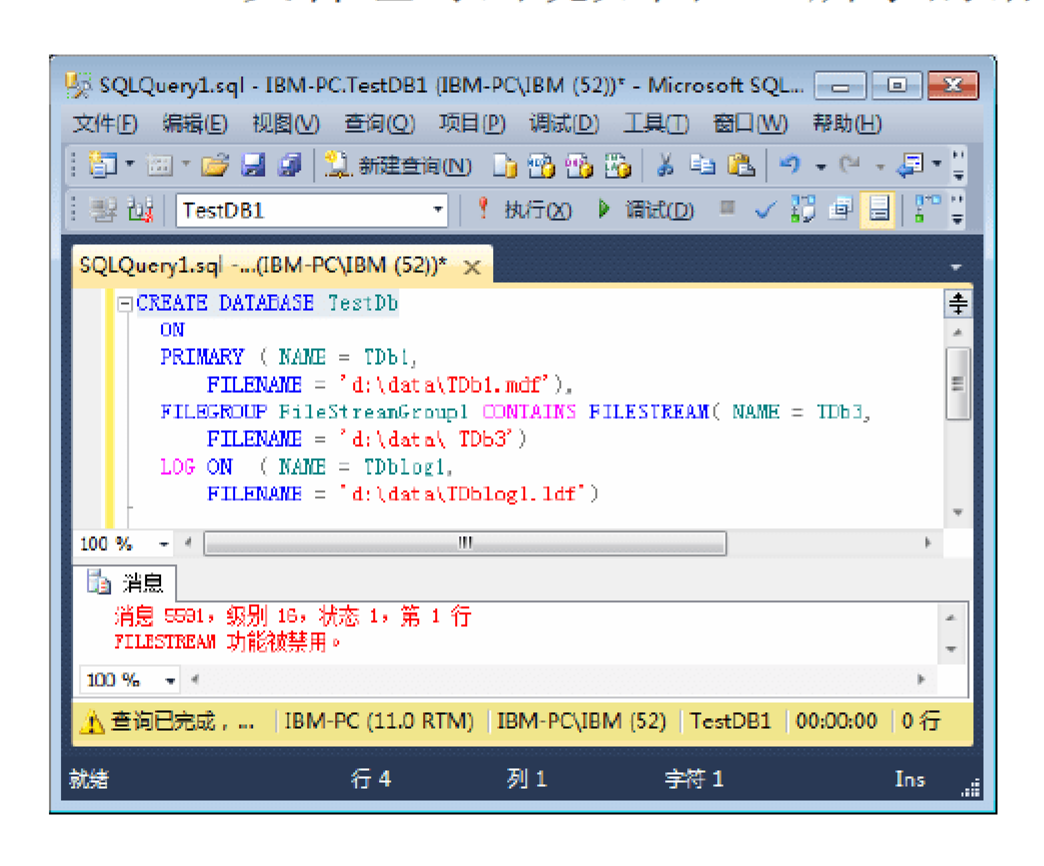


图 4.4　创建文件表失败的错误提示

如果出现了图 4.4 中的错误，就需要在 SQL Server 的配置管理器中找到 SQL Server 的服务，并启用 FILESTREAM 功能，如图 4.5 所示。

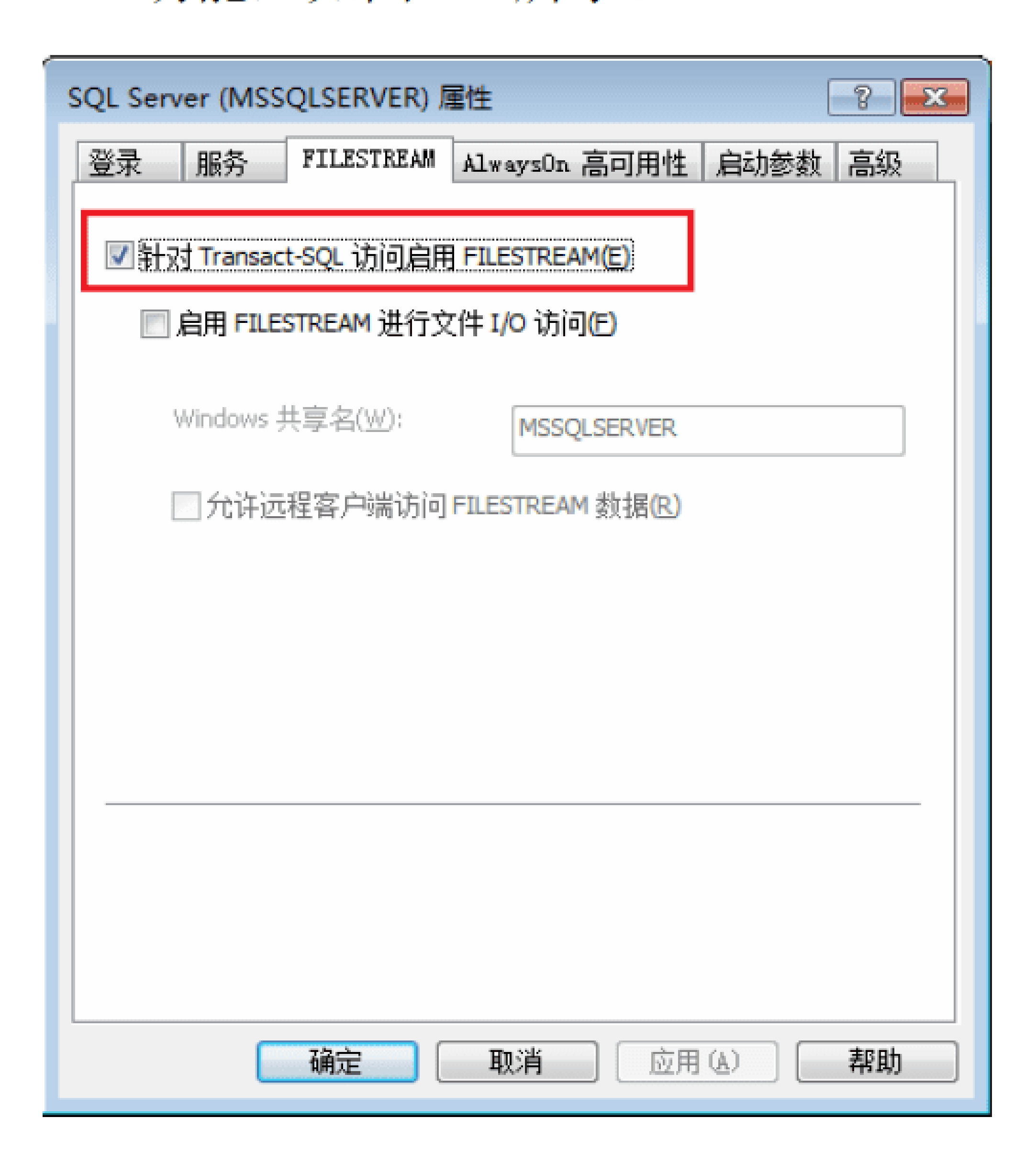


图 4.5　启用 FILESTREAM

完成图4.5的操作后，读者还需要在SQL Server的查询编辑器中执行如下代码更改FILESTREAM的访问级别，如代码4.8所示。

代码 4.8　创建数据库 TestDb

```
EXEC sp configure filestream access level, 2
RECONFIGURE
```

执行了面的代码，就可以顺利地完成数据库的创建了。

2）通过 SSMS 工具来创建文件表

在 SSMS 中创建文件表分为如下两个步骤。

（1）打开新建文件表模板。在对象资源管理器下，依次展开“TestDB 数据库”|“表”，并右击 FileTables 选项，在弹出的菜单中选择“新建 FileTable”选项，如图 4.6 所示。

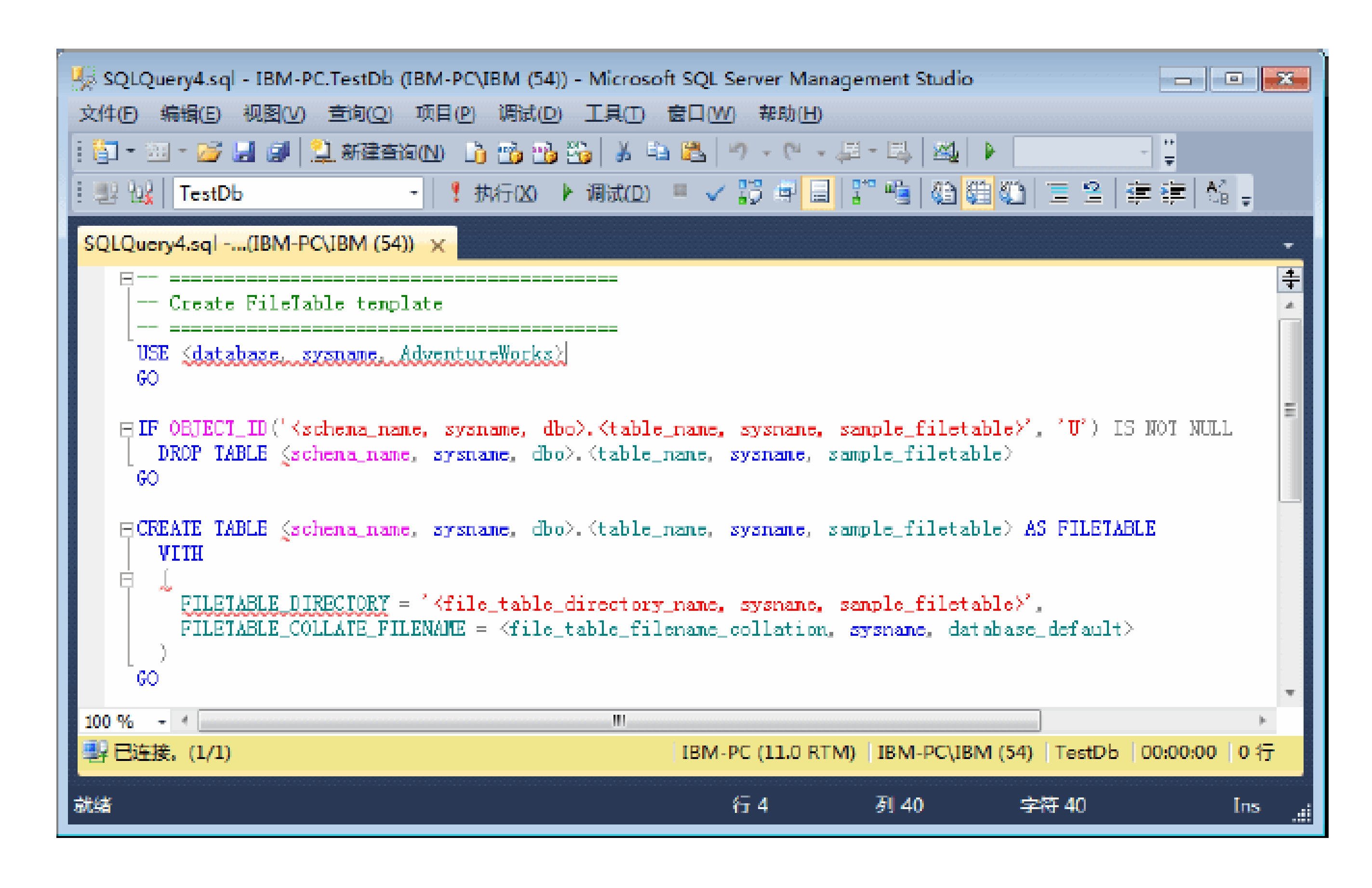


图 4.6　创建文件表模板

（2）填入创建文件表的内容。在图 4.6 所示的界面中，填入创建文件表的正确内容，保存后即可完成文件表的创建操作。

注意：在创建文件表时还需要注意的是，不能将现有的表转换成文件表，不能在系统数据库中创建文件表，并且不能将文件表作为临时表创建。

2．文件表的修改

修改文件表与创建文件表一样，也可以通过语句或者 SSMS 来完成。

（1）使用语句修改文件表

在 SQL Server 2012 中修改文件表只能修改文件表的 FileTable_Directory 选项，具体的修改语句如下所示。

```
ALTER TABLE filetable name
SET(FILETABLE_DIRECTORY = '' );
```

- filetable_name：文件表的名字。
- FILETABLE_DIRECTORY：设置文件表的新的目录名称。

（2）使用 SSMS 修改文件表

在“对象资源管理器”中，右键单击 FileTable 选项，然后选择“属性”，弹出如图 4.7 所示的对话框。

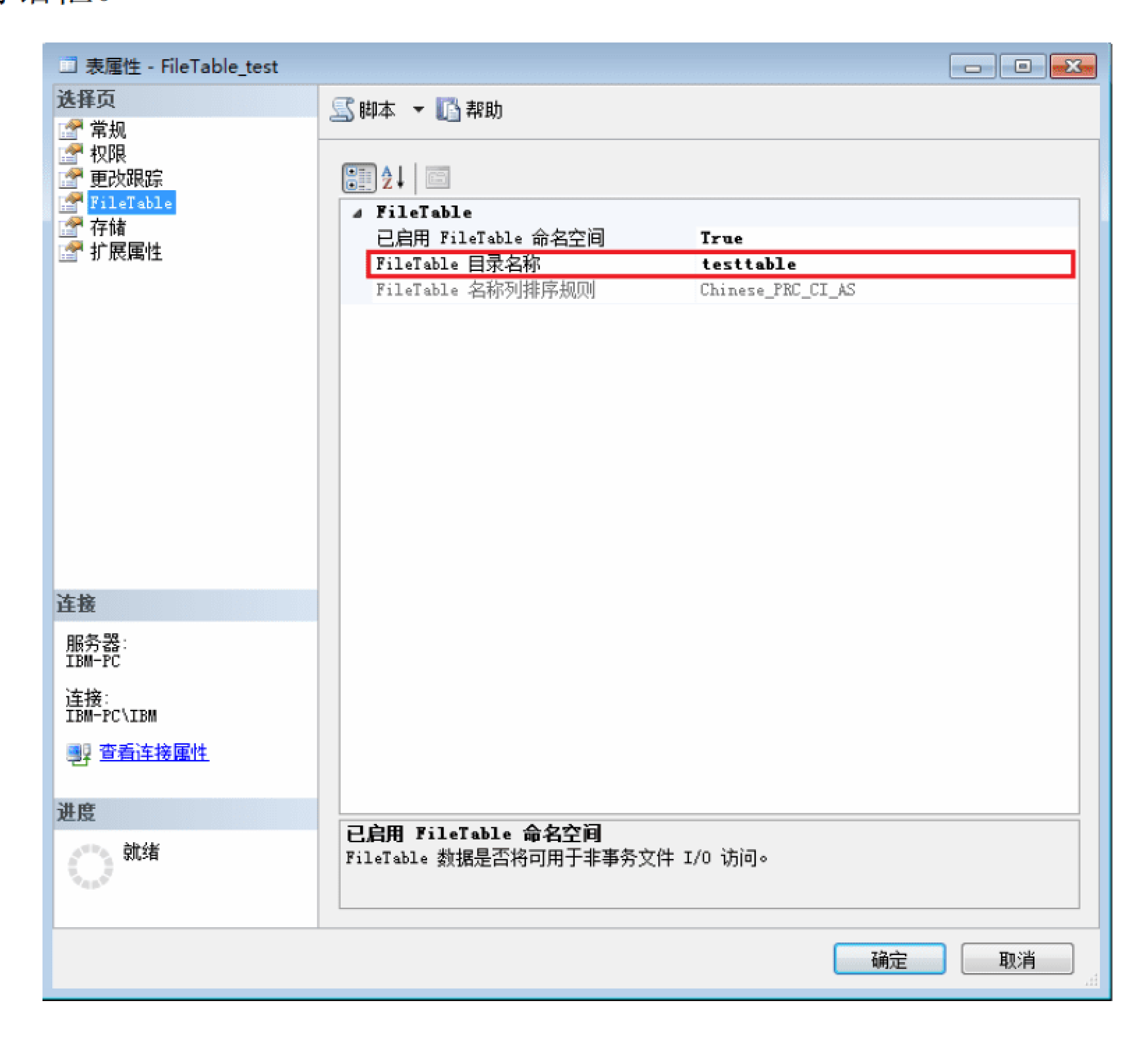


图 4.7　文件表属性对话框

在图 4.7 所示的界面上，在“FileTable 目录名称”文本框中输入新值，单击“确定”按钮即可完成修改文件表操作。

注意：在修改文件表时不能修改 FILETABLE_COLLATE_FILENAME 的值，不能更改、删除或禁用文件表的系统定义的列，也不能将新的用户列、计算列或持久化计算列添加到文件表。

3．删除文件表

删除文件表的操作与删除普通表的操作基本类似，也是通过 DROP TABLE 语句来删除的。此外，同样可以用 SSMS 将文件表选中后直接删除。需要注意的是，删除文件表的同时也会删除如下两类对象。

- 与该表关联的所有对象，如索引、约束和触发器。
- FileTable 目录及其包含的子目录将从数据库的 FILESTREAM 文件组和目录层次结构中删除。

注意：如果 FileTable 的文件命名空间中有打开的文件，那么将无法删除文件表。

4．查询文件表

文件表创建完成后，可以查看文件表信息。查询文件表信息，通常是通过系统视图 sys.filetables 和 sys.filetable_system_defined_objects 来完成的。

- sys.filetables：为数据库中每个文件表返回一行。
- sys.filetable_system_defined_objects：获取在创建关联的 FileTable 时创建的系统定义对象的列表。

4.3.3　其他开发特性

除了在本章中讲解的一些新增加的开发特性外，SQL Server 2012 还引入了一些新的概念和语句，主要有包含的数据库、增强 Windows PowerShell 和语义搜索等功能。

1．包含的数据库

所谓包含的数据库，是指简化将数据库从一个数据库引擎移动到另一个数据库引擎的操作。使用包含的数据库后，包含数据库中的用户就不关联相应的 SQL Server 实例上的登录名了。

2．Windows PowerShell

在以前的 SQL Server 版本的安装程序中提供了 Windows PowerShell 程序，但是从 SQL Server 2012 开始，在其安装程序中就不再提供该程序了。但是，Windows PowerShell 2.0 仍然是安装 SQL Server 2012 的必备组件之一。在安装 SQL Server 2012 之前，读者需要自行安装此软件了。在 SQL Server 2012 中可以使用新的 Windows PowerShell 2.0 功能调用的模块将 SQL Server 组件加载到 PowerShell 环境。通过 sqlps 工具可以启动 PowerShell 2.0，但是目前已经将该工具列入到了不推荐使用的功能列表中，可能在今后的版本中会删除该工具。

3．bcp和sqlcmd实用工具

bcp 工具主要用于在 SQL Server 中导入和导出大容量的数据。SQL Server 2012 在原有的语法基础上添加了-K 选项，连接到服务器时声明应用程序工作负荷类型，唯一可能的值是 ReadOnly。如果未指定-K，bcp 实用工具将不支持连接到 AlwaysOn 可用性组中的辅助副本。

sqlcmd 工具可以在命令提示符处、在 SQLCMD 模式下的“查询编辑器”中、在 Windows 脚本文件中或者在 SQL Server 代理作业的操作系统作业步骤中输入 Transact-SQL 语句、系统过程和脚本文件。在 SQL Server 2012 中对其执行“SELECT * FROM T FOR XML…”时的行为做了一些调整，比如：包含单引号的文本数据不再将其替换成“'”，但它仍为有效的 XML，并且 XML 分析器将提供相同的结果；将没有小数位的 money 数据值显示为 4 位小数。

4．统计语义搜索

统计语义搜索通过提取统计上相关的“关键短语”，然后基于这些短语标识“相似文档”，提供对 SQL Server 数据库中存储的非结构化文档的更深层次剖析。语义搜索以 SQL Server 中现有的全文搜索功能为基础，但允许超出依照句法的关键字搜索范畴的新方案。要使用语义搜索功能首先要安装该功能，可以通过“SELECT SERVERPROPERTY ('IsFullTextInstalled')”语句检查本机是否安装了该功能，如果返回值为 1 表示安装了全文搜索和语义搜索；返回值为 0 表示未安装它们。

5．全文搜索

全文搜索主要有属性搜索和邻近搜索。属性搜索是指通过配置全文索引支持对属性进行属性范围内的搜索。如果要为 varbinary、varbinary(max)、image 或 xml 数据类型的文档建立索引需要进行额外处理，也就是通过筛选器从文档中提取文本。邻近搜索在 SQL Server 2012 开始，可以通过使用 CONTAINS 谓词或 CONTAINSTABLE 函数自定义邻近搜索 NEAR 选项，通过该选项可以指定用来分隔匹配中的第一个和最后一个搜索词的非搜索词的最大数目。此外，在 SQL Server 2012 中还更新了全文搜索和语义搜索使用的所有断字符和词干分析器，不包括韩语。

4.4　商务智能增强

在 SQL Server 2000 时就已经提供了 DTS、分析服务和报表服务，而在 2005 版中对这 3 个功能模块进行了全面的改进，以全面进军商务智能市场。在 SQL Server 2012 中再次对商务智能功能进行了增强。

4.4.1　集成服务增强

集成服务被称为 BI 平台的粘合剂，具有以下几个方面的应用：

- 合并来自异类数据存储区的数据。
- 填充数据仓库和数据集市。
- 清除数据和将数据标准化。
- 将商务智能置入数据转换过程。
- 使管理功能和数据加载自动化。

SQL Server 2005 中设计集成服务的包是在 VS 2005 中进行，SQL Server 2008 的集成服务包则是在 VS 2008 中进行设计，而 SQL Server 2012 的集成服务包则是在 VS 2010 中进行设计的。在 SQL Server 2008 中，集成服务增强了以下功能：

- 执行多数据操作语言操作。在 SQL Server 2008 中支持在 SQL 语句中使用 MERGE 操作。使用 MERGE 操作，可以在一个语句中表达针对特定目标表的多个 INSERT、UPDATE 和 DELETE 操作。目标表基于与源表的联接条件。
- 检索数据源更改的相关数据。INSERT 操作支持将 INSERT、UPDATE、DELETE 或 MERGE 操作的 OUTPUT 子句返回的行插入到目标表中。
- 提高根据表的聚集索引对数据进行排序时的大容量加载操作的性能。OPENROWSET 函数的 BULK 选项支持 ORDER 参数，该参数指定数据文件中的数据如何排序。

SQL Server 2008 中增加了可变更数据捕获功能，通过变更数据捕获功能可以改善增量加载。同时数据集成包现在可以更有效地扩展、使用有效的资源和管理最大的企业级的工作负载。

在集成服务中进行遍历是一种十分常见的 ETL 操作。集成服务增强了对遍历的支持，SQL Server 2008 中遍历可以支持大型的表。

在 SQL Server 2012 中，又在 SQL Server 2008 的基础上对集成服务进行如下增强：

- 在项目连接管理器中可以创建由项目中的多个包共享的连接管理器，可以在 SQL Server Data Tools（SSDT）中来直接操作创建和管理集成服务项目，并在其中使用项目连接管理器。
- 增加了解析列引用编辑器，通过它可以将未映射的输出列与未映射的输入列相链接。此外，该编辑器还将突出显示路径以便指示正在解决的路径。
- 性能方面有所提升。通过增强 Integration Services 合并转换和合并联接转换的强健性和可靠性，能够使合并转换或合并联接转换的包更有效地使用内存。同时，更好地支持了多数据流组件的开发。
- 增加了 DQS 清除转换。在 SQL Server 2012 中，集成服务中引入了数据质量客户端，可以通过该客户端打开集成服务，可以更好地修正数据。DQS 清除转换在满足该列已选定用于数据更正、数据更正支持该列数据类型，以及该列映射到的域具有兼容数据类型条件时处理输入列中的数据。

4.4.2　分析服务增强

分析服务为商务智能应用程序提供联机分析处理（OLAP）和数据挖掘功能。在多维数据处理中，分析服务提供了个性化扩展的功能，开发人员可以创建新的分析服务对象和功能，并在用户会话的上下文中动态提供这些对象和功能。在 SQL Server 2008 中，主要

对分析服务做了如下几个方面的增强。

- 在分析服务的维度设计中，SQL Server 2008 在 BI 设计器上也进行了重大的修改，使用新的属性关系设计器、新的 AMO 警告、经过简化和增强的维度向导，以及新的键列对话框等。在分析服务内部改进了存储结构，使用新的备份和还原功能，从而改善了分析服务的性能。备份和还原分析服务数据库的功能也得到增强，减少了对数据库大小的限制，备份和还原操作需要的时间也大大降低。
- 在数据挖掘方向上，分析服务为了改进时序模型中某些预测的准确性和稳定性，在 Microsoft 时序算法中增加了一种新的算法。该新算法基于熟知的 ARIMA 算法，它比分析服务中一直使用的 ARTxp 算法可提供更佳的长期预测。

SQL Server 2008 中的分析服务还提供了对挖掘结构的重要改进，包括筛选外部数据集和模型事例、使用别名、查询挖掘结构，以及从挖掘模型到基础挖掘结构的钻取。

在 SQL Server 2012 中，分析服务部分也做了部分功能的增强，同时引入了表格建模的方法。在安装 SQL Server 2012 时，需要选择要安装的分析服务的类型，主要有 3 个类型，即多维和数据挖掘模式、表格模式，以及 PowerPivot for SharePoint 模式，默认安装的是多维和数据挖掘模式。如果读者想尝试使用表格模式和 PowerPivot for SharePoint 模式请务必在安装数据库时进行安装，否则只能重新安装分析服务。SQL Server 2012 中分析服务部分主要的新增功能如下所述。

- 表格模型：它主要是应用在面向表格的数据中，当用结果集访问表列数据时用该模式的效果较其他模型更好。目前表格模型已经与 SQL Server Data Tools 工具集成，可以在该工具中直接创建表格模型。同时，也可以在 SSMS 中对表格数据库进行管理。在表格模型中也可以设计视图、角色、设置行级的安全性，并且在表格模型中取消了每个表 20 亿行的上限，对表中行数没有限制。但是，表中的列被限制为最多 20 亿个非重复值。表格模型中的内存分页可以大于服务器的物理内存。另外，还在 DAX 功能中引入了新的统计函数、表函数、搜索函数、行级安全性函数等多种函数。
- 多维模式：在该版本中取消了对字符串存储文件 4GB 的显示，允许文件根据需要增加。添加了 Resource Usage 是一个新的事件类，可以用于处理查询时的资源使用情况。引入了 Locks Acquired、Locks Released 和 Locks Waiting 3 个新的跟踪事件，并完善了现有的锁事件 Deadlock 和 LockTimeOut，使之能更好处理与锁相关的查询或处理的操作。

此外，还引入了 PowerPivot 的配置工具，并且 PowerPivot 工作簿还可以自动升级以启用数据刷新。

4.4.3　报表服务增强

报表服务是在 SQL Server 2000 SP3 时集成到 SQL Server 中的，用于生成从多种关系数据源和多维数据源中提取内容的企业报表，发布能以各种格式查看的报表，以及集中管理安全性和订阅。

报表服务包含用于创建和发布报表及报表模型的图形工具和向导、用于管理报表服务的报表服务器管理工具，以及用于对报表服务对象模型进行编程和扩展的应用程序编程接

口（API）。SQL Server 2008 中对报表服务中的报表设计和核心都进行了增强。

报表服务中内置的表单身份验证允许用户在 Windows 身份验证与表单身份验证两种模式间轻松的切换。

除了在 BI 开发平台设计报表外，微软还专门提供了一款独立的报表设计器 Microsoft SQL Server 2008 Report Builder，通过报表设计器可以轻松地创建任意结构的即时权属报表。如图 4.8 所示为报表设计器中创建的一个新报表的界面。

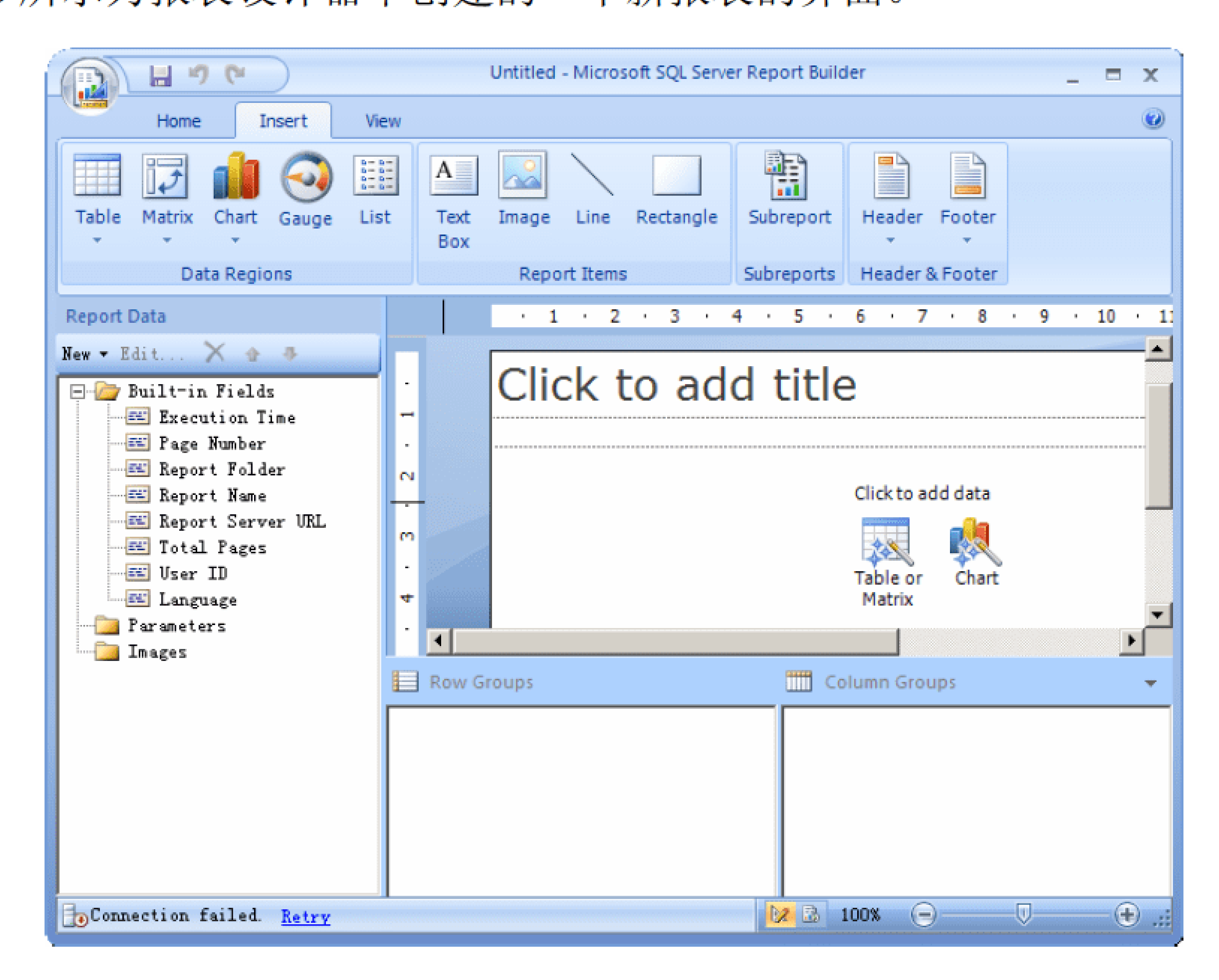


图 4.8　报表设计器

注意：SQL Server 2008 中报表服务器应用内嵌，允许报表内 URL 及其订阅在前端应用显示其后台地址。

在 SQL Server 2012 中，报表服务中也有很多的改进和新增加的功能，主要的新特性如下所述。

- Power View：它是一种从 SharePoint Server 2010 内启动的基于浏览器的 Silverlight 应用程序，使用户能够通过交互式的展示展现数据中的内在关系并与组织中的他人分享。使用 Power View 的报表，拥有图块、切片器、图表筛选器等多种可视化对象，便于查看和分析报表。
- 数据报警：它是一种数据驱动的警报解决方案，如果报表数据中发生了令用户感兴趣的更改，它会在恰当的时间通知该用户。数据报警功能提供了警报的定义和保存、运行，以及向收件人发送警报的基本操作。同时，为不同的用户提供了不同的操作工具，有针对用户的数据警报管理器，也有针对管理员的数据警报管理器。
- 与 VS 2010 集成：在 SQL Server 2012 中，SQL Server Data Tools 是 Visual Studio 2010 的外接程序。因此，可以在 SQL Server Data Tools 中直接创建报表项目，并可以将 SQL Server 2008 的报表项目自动升级到 SQL Server 2012 的项目中。

此外，还在 SharePoint 模式方面有了一定的改进，比如支持 SharePoint 备份和恢复，以及端到端 SharePoint ULS 日志记录等方面。

4.4.4　Office 集成

SQL Server 2008 中将商务智能功能与 Office 集成，提供了一种全新的 Word 呈现，允许用户直接从微软 Office Word 中显示报表。另外，现有的 Excel 2007 的功能已经大大增强了，以便适应对新特性的支持，如嵌套数据区域、子报表以及合并单元格改进。这让用户能精确维护布局，并提高来自微软 Office 应用中报表的总消费量。

SQL Server 2008 中的分析服务也与 Excel 和 Visio 进行了集成，通过下载使用 Office 2007 数据挖掘外接程序（数据挖掘外接程序），可以在 Office Excel 2007 和 Office Visio 2007 中利用 SQL Server 2008 的预测分析功能。分析服务 Office 集成中包括以下组件。

- Excel 表分析工具：此外接程序提供了一些非常易于使用的任务，这些任务可利用 SQL Server 2008 数据挖掘对电子表格数据进行更强大的分析，而不需要用户了解任何数据挖掘概念。
- Excel 数据挖掘客户端：通过使用此外接程序，用户可以通过使用电子表格数据或通过 SQL Server 2008 Analysis Services 实例访问的外部数据，在 Excel 2007 内创建、测试、浏览和管理数据挖掘模型。
- Visio 数据挖掘模板：利用此外接程序，可以用可加注的 Visio 2007 绘图形式呈现和共享挖掘模型。

除了与 Office 客户端进行集成外，SQL Server 2008 的报表服务还与 Microsoft Office SharePoint Server 2007（简称 MOSS）进行集成。使用报表服务与 MOSS 进行集成具有以下几个优点。

- 使用一个一致的用户接口来管理和查看报表。
- 使用 MOSS 中的版本和报表的工作流存储到 MOSS 文档库中时跟踪报表。
- 通过 MOSS 文档库管理一个单独的用于报表的安全模型。
- 使用 MOSS 即开即用报表中心模板轻松地建立一个站点用于存储报表。

另外 Office 服务器产品中还有个 Microsoft Office PerformancePoint Server，其是一个集成的性能管理应用，SQL Server 2008 也与该产品进行了集成。用户可以使用它来监控、分析和计划基于 SQL Server 2008 提供的分析数据的商业活动。

在 SQL Server 2012 中，也在 Office 集成方面做了一定的工作，主要体现在 Excel 呈现器和 Word 呈现器。

- Excel 呈现器：SQL Server 2012 中新增的 Reporting Services Excel 呈现扩展插件可将报表呈现为能够兼容 Microsoft Excel 2007～2010 及 Microsoft Excel 2003 的 Excel 文档。并且消除了之前版本的限制，与 Excel 2003 兼容。目前的工作表的最大行数是 1 048 576 行，最大列数是 16 384 列，允许 24 位颜色。
- Word 呈现器：与 Excel 呈现器一样，也可以兼容 Microsoft Word 2007～2010 及 Microsoft Word 2003 的 Word 文档。同时，通过使用 Word 呈现器导出的报表通常显著小于通过使用 Word 2003 呈现器导出的相同报表。

4.4.5　数据质量分析

在 SQL Server 2012 中新增加了一个数据质量分析（Data Quality Services）的服务功能。该功能可以维护数据的质量并确保数据满足业务使用的要求。数据质量分析是一种知识驱动型解决方案，该解决方案通过计算机辅助方式和交互方式来管理数据源的完整性和质量。使用该功能可以发现、生成和管理有关数据的知识，然后可以使用该知识执行数据清理、匹配和事件探查。下面就从以下几个方面说明数据质量分析服务提供的常用功能。

- 数据清理：是在数据源中分析数据质量的过程，在其中手动批准/拒绝系统的建议并将对数据进行更改。通常使用计算机辅助方式和交互方式两种方式共同完成。如果要使用计算机辅助方式清理数据，还需要创建数据质量项目用以清理数据，并指定要清理的数据表、视图或 Excel 数据表，运行计算机辅助清理程序后可以得到要用于交互式清理过程的数据质量信息。在计算机辅助清理结束后，就可以使用交互式清理对数据按照不同的建议进行清理。在完成清理工作后，数据可以导出到 SQL Server 数据库中的新表、Excel 文件等形式供用户使用。
- 匹配：它可以分析单个数据源的所有记录中的重复程度，同时返回所比较的每组记录之间的加权概率。数据质量分析中的匹配操作是通过建立匹配策略，并运行匹配项目，最后将相应的匹配结果和过程导出到 SQL Server 数据数据表或.csv 文件中。
- 事件探查：它是一个分析现有数据源中的数据并显示有关数据质量分析活动中的数据统计信息的过程。它提供的统计信息表明用户正在源数据的知识管理或数据质量项目中执行的特定操作的效果。
- 监视：通过监视可以跟踪和确定数据质量活动的状态。该功能只要是在数据质量分析的客户端中完成的，但是该监视不能监视系统级别的活动。

此外，数据质量分析中还提供了知识库，用于在用户创建数据质量过程中不断增强与用户有关的数据的知识，从而不断提高数据质量。

4.5　小　　结

本章主要集中介绍了 SQL Server 2012 的特性。SQL Server 2012 中增强了 SSMS 的功能，使得用户在进行开发和管理时变得更方便、更全面。SQL Server 2012 同时提供新增的空间数据类型和系统视图，还将 T-SQL 语句进行了加强，增加了多个 SQL 语法功能。此外，SQL Server 2012 还提供了列存储索引和文件表等新的开发特性。另外，在商务智能方面也进行了加强，特别是提供了数据质量分析服务，使用户可以更好地保证数据源的正确性。

第 2 篇　数据库安全

第 5 章　SQL Server 2012 安全

随着网络时代的到来，越来越多的业务系统运行在互联网上，越来越多的数据都以比特的方式保存在硬盘上，数据的安全成了人们日益关注的一个问题。作为一款强大的企业级数据库管理系统，SQL Server 在安全方面做了很多的工作。本章将主要讲解 SQL Server 2012 在安全方面的特性。

5.1　新安全机制概论

可将保护 SQL Server 视为一系列步骤，它涉及 4 方面：平台、身份验证、对象（包括数据）及访问系统的应用程序。本节将从这 4 个方面进行简单的介绍。

5.1.1　平台与网络安全性

SQL Server 的平台包括物理硬件和将客户端连接到数据库服务器的联网系统，以及用于处理数据库请求的二进制文件。

物理安全性的最佳实践是严格限制对物理服务器和硬件组件的接触。例如，将数据库服务器硬件和联网设备放在限制进入的上锁房间。此外，还可通过将备份媒体存储在安全的现场外位置，限制对其接触。实现物理网络安全首先要防止未经授权的用户访问网络。具体实现包括以下几点。

1．关闭不必要的网络协议

在第 1 章讲解 SQL Server 2012 的协议中已经讲到，SQL Server 2012 支持 4 种不同的协议进行连接。如果资源有限，SQL Server 与应用程序是在同一台服务器上（当然，这是一种不推荐的做法），此时就可以只启用 SQL Server 的共享内存协议，而将其他协议关闭。只开启共享内存协议后，没有任何人可以通过网络直接连接 SQL Server，当然也就提高了网络安全性。

如果 SQL Server 服务器和应用程序服务器是在同一个局域网中（例如服务器之间使用网线直接连接）如图 5.1 所示，则对外暴露的只有应用程序服务器，数据库服务器与应用程序服务器在同一个局域网中，此时只需开启命名管道协议即可。

2．指定并限制用于SQL Server的端口

由于网络架构原因或其他因素需要将 SQL Server 服务器直接暴露在因特网上，如果使用默认的 SQL Server 端口 1433 将是十分危险的。黑客通过端口扫描便可断定扫描到的服

务器是数据库服务器，通过利用系统漏洞、SQL 注入或利用木马病毒等方法获得了数据库的密码后便可直接通过因特网连接数据库服务器，进行破坏。提高因特网上数据库服务器安全的比较实用的方法就是修改 SQL Server 的连接端口。

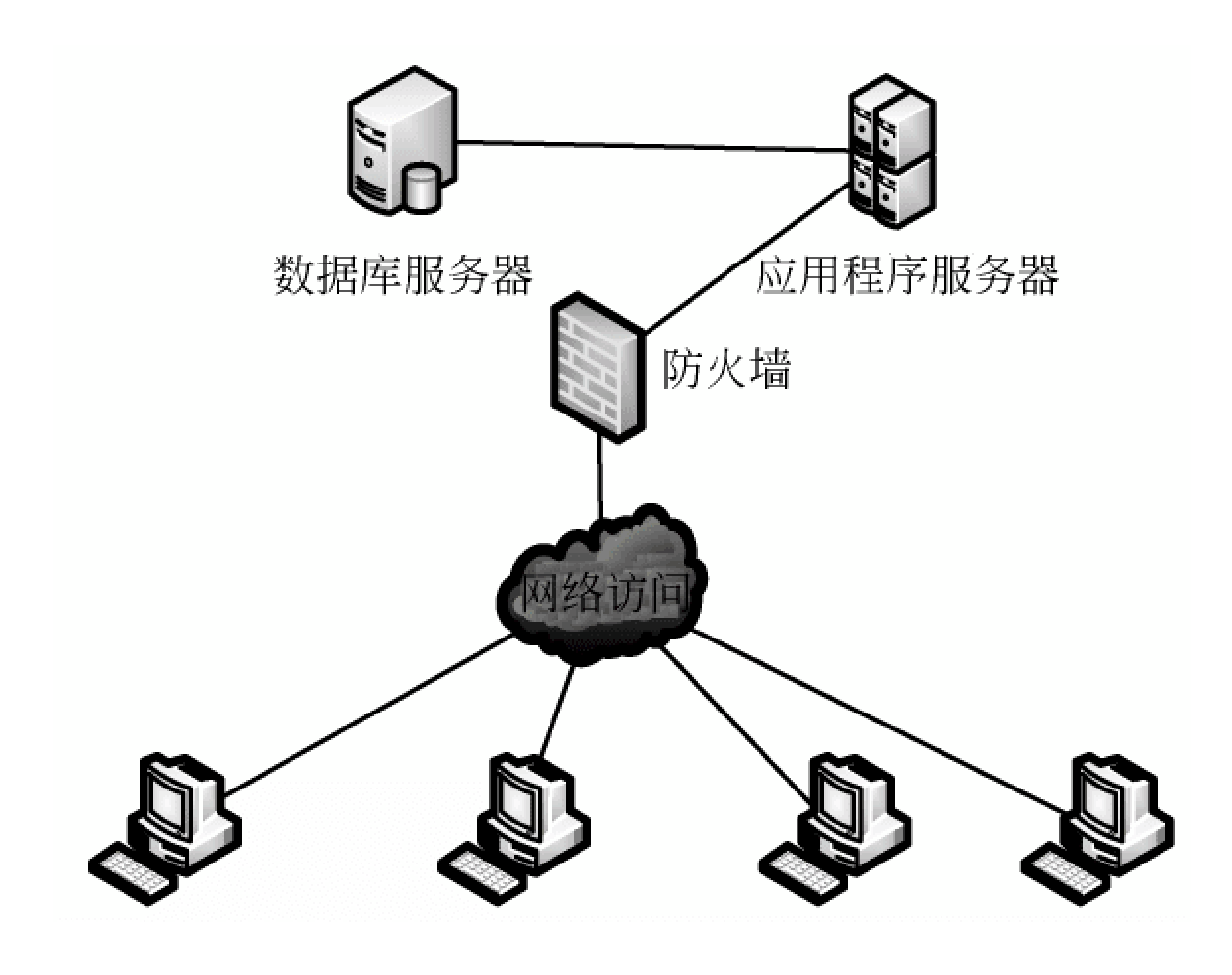


图 5.1　应用程序服务器直连数据库服务器

使用 SQL Server 配置管理工具可以修改 TCP/IP 协议中使用的端口，将端口改为比较陌生的端口，例如 8412，或者修改为其他服务的端口例如 443（主要是用于 HTTPS 服务）都可以误导入侵者，提高系统的安全性。

3．限制对SQL Server的网络访问

仅仅是对默认端口的修改并不能保证数据库的安全，在黑客得知某端口对应的是数据库服务后仍然可以通过各种手段对数据库资料进行窃取和破坏。对于暴露在因特网上的服务器，都应该使用防火墙来限制网络的访问，提高系统的安全。实际上不仅仅是针对因特网，即使是局域网也有必要使用防火墙来提高系统的安全性。

如图 5.2 所示为一般大中型企业的服务器网络架构，在企业应用中充斥着各种应用服务器，各种服务器上运行着不同的服务，虽然大部分服务器是统一存放在一个机房中，但是在网络上它们之间是互不相连的。所有的服务器都是通过防火墙再与其他系统或用户进行交互。在防火墙上便可以配置服务器之间的网络访问策略。

通过使用防火墙限制对 SQL Server 的网络访问可以很大限度地提高数据库的安全性。例如在图 5.2 中的网络架构中，如果数据库只用于网站应用，那么在防火墙上便可以设定只有 Web 服务器能够访问数据库服务器。这样即使黑客知道了数据库服务器的地址，知道了开放的端口，哪怕知道了数据库的密码也无法通过网络非法访问数据库服务器。

服务器之间互不相连可以防止一台服务器被攻陷后，黑客以该服务器为肉鸡（黑客用语，即当做跳板的机器）轻易攻陷其他服务器。例如黑客通过某系统漏洞攻陷了电子邮件服务器，但是由于防火墙限制了电子邮件服务器对数据库服务器的访问，所以数据库服务器仍然是安全的。

4．备份和还原策略

在大中型企业应用中，数据库文件通常保存在 SAN 或磁盘阵列上，这样使数据库文件

丢失或损坏的几率大大下降，但是为了减少人为误操作或恶意破坏造成的损失，日常对数据库的备份必不可少。

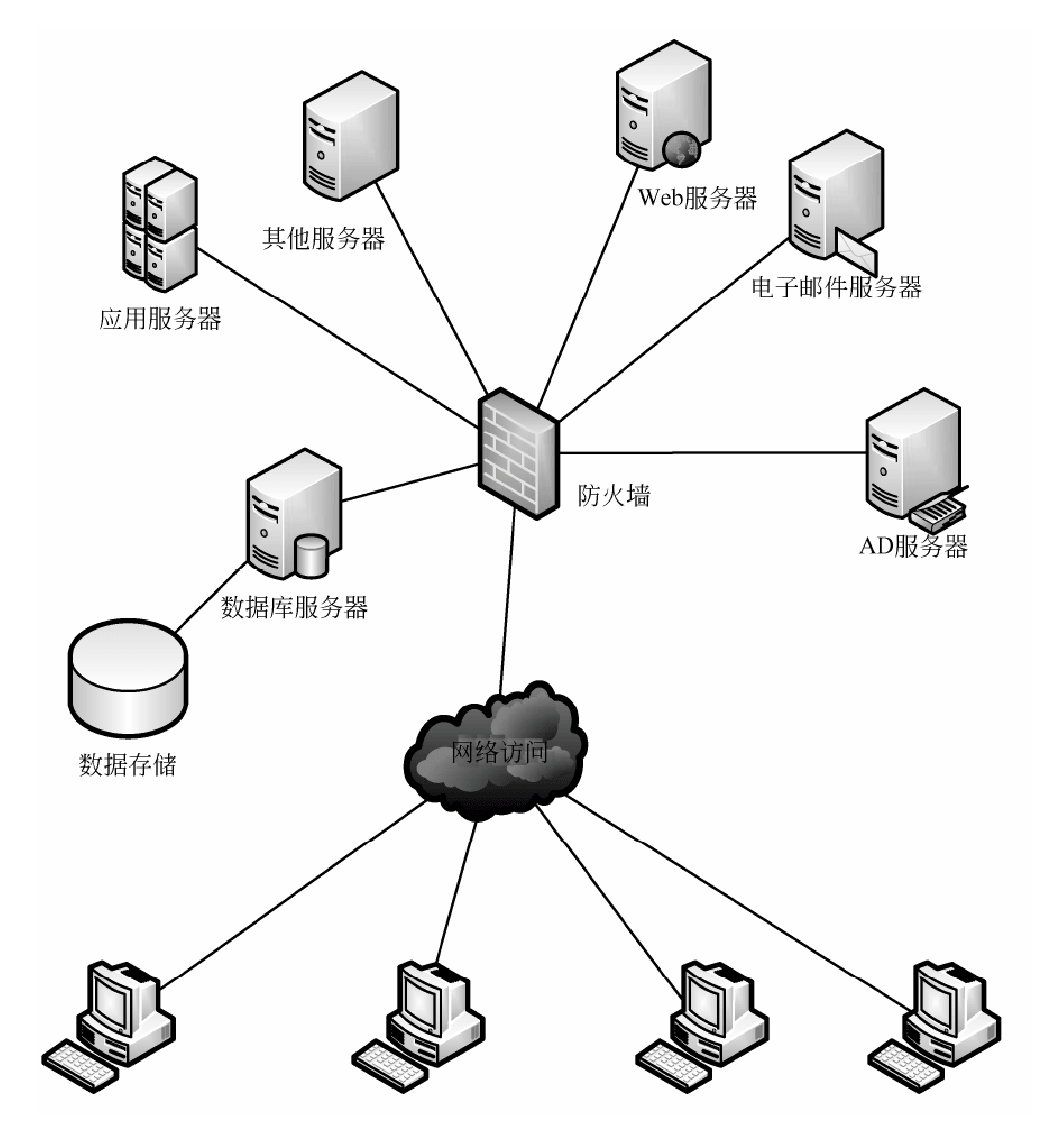


图 5.2　一般大中企业服务器网络架构

对数据库应该尽早而且经常备份。如果数据库每天备份，那么即使发生意外，损失顶多是一天的数据；而如果数据库是每周备份一次，则有可能会损失一周的数据。

对数据库备份并不是将数据库备份到相同的磁盘上并忘记它。数据库备份应该存放在一个独立的位置（最好是远离现场），以确保其安全，要不然一旦存储发生意外，数据库文件和备份就一同被损坏了。在大型企业中一般采用磁带机进行备份并将备份的磁带专门存储在一个安全的地方。

除了物理上和网络上的安全外，操作系统安全性也至关重要。及时更新操作系统 Service Pack 和升级包含重要的安全性增强功能。通过数据库应用程序对所有更新和升级进行测试后，再将它们应用到操作系统。

减少外围应用是一项安全措施，它涉及停止或禁用未使用的组件。减少外围应用后，对系统带来潜在攻击的途径也会减少，从而有助于提高安全性。限制 SQL Server 外围应用的关键在于通过仅向服务和用户授予适当的权限来运行具有“最小权限”的所需服务。

5.1.2　主体与数据库对象安全性

主体是指获得了 SQL Server 访问权限的个体、组和进程。“安全对象”是服务器、数据库和数据库包含的对象。每个安全对象都拥有一组权限，可对这些权限进行配置以减少 SQL Server 外围应用。

安全对象是 SQL Server 数据库引擎授权系统控制对其进行访问的资源。通过创建可以为自己设置安全性名为“范围”的嵌套层次结构，可以将某些安全对象包含在其他安全对象中。安全对象范围有服务器、数据库和架构，分别包括以下数据库对象。

- 服务器安全对象包括：端点、登录账户和数据库。
- 数据库安全对象包括：用户、角色、应用程序角色、程序集、消息类型、路由、服务、远程服务绑定、全文目录、证书、非对称密钥、对称密钥、约定和架构。
- 架构安全对象包括：类型、XML 架构集合和对象。这里的对象包括聚合、约束、函数、过程、队列、统计信息、同义词、表和视图。

SQL Server 支持安全套接字层（SSL），并且与 Internet 协议安全（IPSec）兼容。启用 SSL 加密，将增强在 SQL Server 实例与应用程序之间通过网络传输的数据安全性。但是，启用加密的确会降低性能。SQL Server 与客户端应用程序之间的所有通信流量都使用 SSL 加密时，还需要进行以下额外处理：

- 由于加密机制的需要，连接时需要进行额外的网络往返。
- 从应用程序发送到 SQL Server 实例的数据包必须由客户端网络库加密并由服务器端网络库解密。
- 从 SQL Server 实例发送到应用程序的数据包必须由服务器端网络库加密并由客户端网络库解密。

除了 SQL Server 连接加密以外，SQL Server 还提供了大量函数支持加密、解密、数字签名，以及数字签名验证。加密并不解决访问控制问题。不过，它可以通过限制数据丢失来增强安全性，即使在访问控制失效的罕见情况下也能如此。例如，在数据库主机配置有误且恶意用户获取了包含敏感数据（如信用卡号）数据库的情况下，如果被盗信息已加密，则此信息将毫无用处。

在 SQL Server 加密中，证书是在两个服务器之间共享的软件“密钥”，使用证书进行加密后，可以通过严格的身份验证实现安全通信。可以在 SQL Server 中创建和使用证书，以增强对象和连接的安全性。

5.1.3　应用程序安全性

实际与用户交流的是应用程序客户端，所以 SQL Server 安全性包括编写安全客户端应用程序。不安全的客户端应用程序容易出现 SQL 注入漏洞或暴露数据库链接信息。在最简单的情况下，SQL Server 客户端可与 SQL Server 实例运行在同一台计算机上。对于一般的企业应用来说，通常一个客户端可能会通过网络连接多个服务器，而一个数据库服务器也可能连接着多个客户端。默认的客户端配置可以满足大多数情况。

SQL Server 客户端可以使用多种方式来连接数据库。一般是 SQL Server Native Client

OLE DB 访问接口连接到 SQL Server 实例。使用 ADO.NET 编程连接 SQL Server，以及 sqlcmd 命令提示工具和数据库管理工具 SQL Server Management Studio，都是 OLE DB 应用程序的例子。另外，随 SQL Server 旧版本安装的客户端实用工具则使用 SQL Server Native Client ODBC 驱动程序连接到 SQL Server。除了 OLE DB 和 ODBC 方式外，有些程序是使用 DB-Library 的客户端连接数据库。不过 SQL Server 对使用 DB-Library 的客户端支持，仅限于 Microsoft SQL Server 7.0 功能。

无论客户端使用哪种方式连接 SQL Server，都应该根据实际项目要求对客户端进行管理。对客户端管理的范围可以小到输入服务器名称，大到生成自定义配置项库，以便满足各种各样的多服务器环境。

5.2 账号管理

SQL Server 2012 账号管理分为登录验证、权限、角色和架构等。通过对账号的管理可以有效地提高数据库系统的安全，降低维护的成本。本节将对 SQL Server 账号管理进行讲解。

5.2.1 安全验证方式

在数据库的使用中，经常看到一个数据库中的超级管理员用户（sa）被整个部门甚至整个公司的每一个员工使用，而整个数据库中可能就只有这一个账户，也就是说每个人都知道超级管理员的密码。这样是非常不安全的，因为并不知道到底是谁登录了系统。

数据库的超级管理员应该限制在几个人（如 DBA）之内，超级管理员具有完全的访问权限。对于不同的部门或者项目组的用户，使用不同的用户，而且应该遵循权限最小化的原则，即只提供相应数据库中需要使用的权限，其他权限都不提供。最理想的情况是每个账号只对应一个人，而且只有他本人知道该账号的密码，这样就可以知道谁登录了数据库，做了什么操作了。

SQL Server 2012 提供了两种身份认证方式：Windows 身份认证和 SQL Server 身份认证。

Windows 身份认证是基于 Windows 操作系统自身的身份认证方式进行的安全验证。Windows 身份认证中使用的用户包括本地用户和活动目录（AD）上的域用户。

对于使用 AD 进行管理的企业和开发团队来说，使用 Windows 身份认证是一种比较方便有效的管理办法。SQL Server 并不管理域用户的密码，域用户统一由 AD 服务器负责管理。可以在 SQL Server 中为每个域用户配置相应的权限，而使用了域用户的客户端在登录数据库时也跟 Windows 本地登录一样，不用再输用户名密码即可登录。

SQL Server 身份认证是由 SQL Server 系统自身维护的一套用户系统。在没有域的情况下就需要使用 SQL Server 身份认证进行远程登录。默认情况下，如果安装 SQL Server 时选择了混合身份认证选项，则 sa 账户是系统默认的全局超级管理员。使用 SQL Server 身份认证具有以下优点：

- 用户不一定是域账户也可以远程访问系统。

- 很容易用程序控制用户信息。
- 比基于 Windows 认证的安全性更容易维护。

5.2.2　密码策略

SQL Server 自身并不设置密码策略，SQL Server 通过 Windows 操作系统来实施密码策略。Windows 的密码策略包括：密码复杂性、密码长度最小值、密码最长使用期限、密码最短使用期限、强制密码历史和可还原的密码加密存储密码。

要查看当前 Windows 的密码策略或修改密码策略，可以使用 Windows 自带的管理工具“本地安全策略”，在开始按钮下的管理工具中可以找到该选项。打开“本地安全策略”对话框后，展开左侧“安全设置”选项下的“账户策略”选项，选中“密码策略”便可看到当前系统的密码策略，如图 5.3 所示。

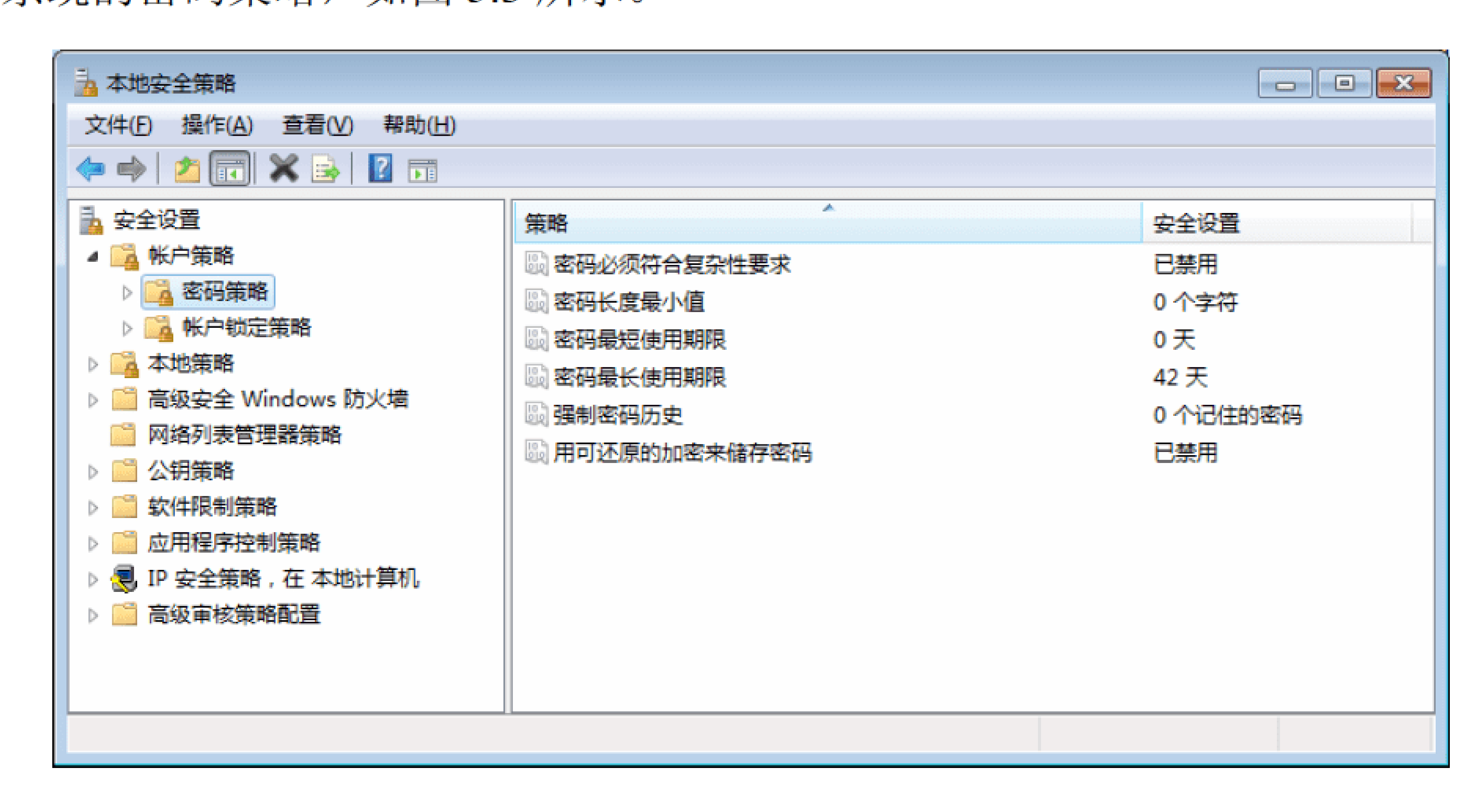


图 5.3　“本地安全策略”对话框

如果启用“密码必须符合复杂性要求”策略，则密码必须符合以下最低要求：

- 不得明显包含用户账户名或用户全名的一部分。
- 长度至少为 6 个字符。
- 包含来自以下 4 个类别中的 3 个字符：英文大写字母（从 A～Z）、英文小写字母（从 a～z）、10 个基本数字（从 0～9）和非字母字符（例如，!、$、#、%）。

人们一般习惯于使用个人的数字（生日、手机号和证件号等）、名字、易记数字（123456 或 888888 等）和其他个人信息作为密码，如果没有启用密码策略，这些密码将很容易通过字典穷举的方式被破解。而启用密码策略后这些低强度的密码将不能作为合法密码使用，为了便于记忆仍然可以使用这些常见的密码进行组合变形后使用，例如 P@ssw0rd!、Study(a)163.com、HeHuan0915!等易记却不易被破解。

密码长度最小值可用于进一步加强密码的强度，在密码复杂性中要求密码至少有 6 个字符，这里可以设置为 10 个或更多来进一步提高密码强度。

密码最长使用期限用于设置密码的有效期。在有效期之后原密码将无效，用户必须更改为新的密码才能登录系统。设置密码有效期是为了防止密码被别人长期使用或密码泄露。

例如由于设置的密码比较复杂，不容易记住，所以计算机的管理员 A 将密码写在了笔记本上，几个月后该笔记本被 B 获得，如果设置了密码有效期，A 在这之前修改了密码，则 B 得到的只是几个月前的密码，而不是系统的密码，从而保证了系统的安全。

说明：密码最短使用期限用于设置两次修改密码的时间间隔，为 0 表示可以随时修改密码。

强制密码历史是强制修改后的密码与前几次密码必须不相同，这是为了防止用户在密码过期后为了一时方便将新密码设置为和前几次的密码中的一个相同。若需要强制密码历史笔者建议至少跟踪老口令 10 次，不能让用户在这一时期内使用相同的密码两次。

在设置了 Windows 的密码策略后，在 SQL Server 中建立登录用户时可以选择强制实施密码策略。关于如何创建用户和如何应用密码策略将在 5.3 节进行讲解。

5.2.3　高级安全性

guest 账户提供了具有默认访问权限的一种方法。当 guest 账户被激活时，登录用户获得了没有直接给他们提供访问权限的数据库的 guest 级访问权限。同时外面的用户可以使用 guest 账户登录得到访问权限。在 SQL Server 中有必要对 guest 账户进行处理，删除该账户可能访问的机会。

除了 guest 账户外 sa 账户是 SQL Server 中最敏感的账户了，有 SQL Server 常识的人都知道，sa 账户是超级管理员账户，在知道了用户名的情况下只需要破解密码比同时破解用户名和密码简单得多。出于安全的考虑，最好能建立其他的登录用户来代替 sa，而将 sa 禁用。如果是使用 Windows 账户登录，则系统默认是禁用了 sa 的。

另外，在 SQL Server 中有一个非常特殊的存储过程 xp_cmdshell，运行该存储过程需要具有对应的权限。该存储过程用于生成 Windows 命令 shell，并以字符串的形式传递以便执行。该存储过程常被用于提升权限等非法操作，在 SQL Server 2000 中该存储过程是启用的，但是在 SQL Server 2012 中默认禁用了该存储过程。读者若想知道当前系统是否启用了 xp_cmdshell，只需要运行以下命令即可。

```
EXEC master..xp_cmdshell 'dir'
```

如果未启用该存储过程，系统将会抛出异常：

```
SQL Server 阻止了对组件'xp cmdshell'的过程'sys.xp cmdshell'的访问，因为此组件已作为此服务器安全配置的一部分而被关闭……
```

如果读者需要启用 xp_cmdshell，可以使用 sp_configure 来启用。具体启用脚本如代码 5.1 所示。

代码 5.1　启用 xp_cmdshell

```
EXEC sp_configure 'show advanced options', 1 --修改服务器配置
GO
RECONFIGURE
GO
EXEC sp_configure 'xp_cmdshell', 1
```

```
GO
RECONFIGURE
GO
```

在 SQL Server 2005 版中除了使用 SQL 命令外也可以通过 SQL Server 自带的 SQL Server 外围应用配置器来启用或禁用 xp_cmdshell，但是在 2012 版中该工具被取消。启用 SQL Server 外围应用配置器后，单击“功能的外围应用配置器”链接，系统将打开功能的外围应用配置器窗口，在按实例查看选项卡中可以找到 xp_cmdshell，单击 xp_cmdshell，右侧将出现“启用 xp_cmdshell”的复选框，如图 5.4 所示。若要启用则选中该复选框，反之则不选中，单击“确定”按钮即可完成对 xp_cmdshell 的配置。

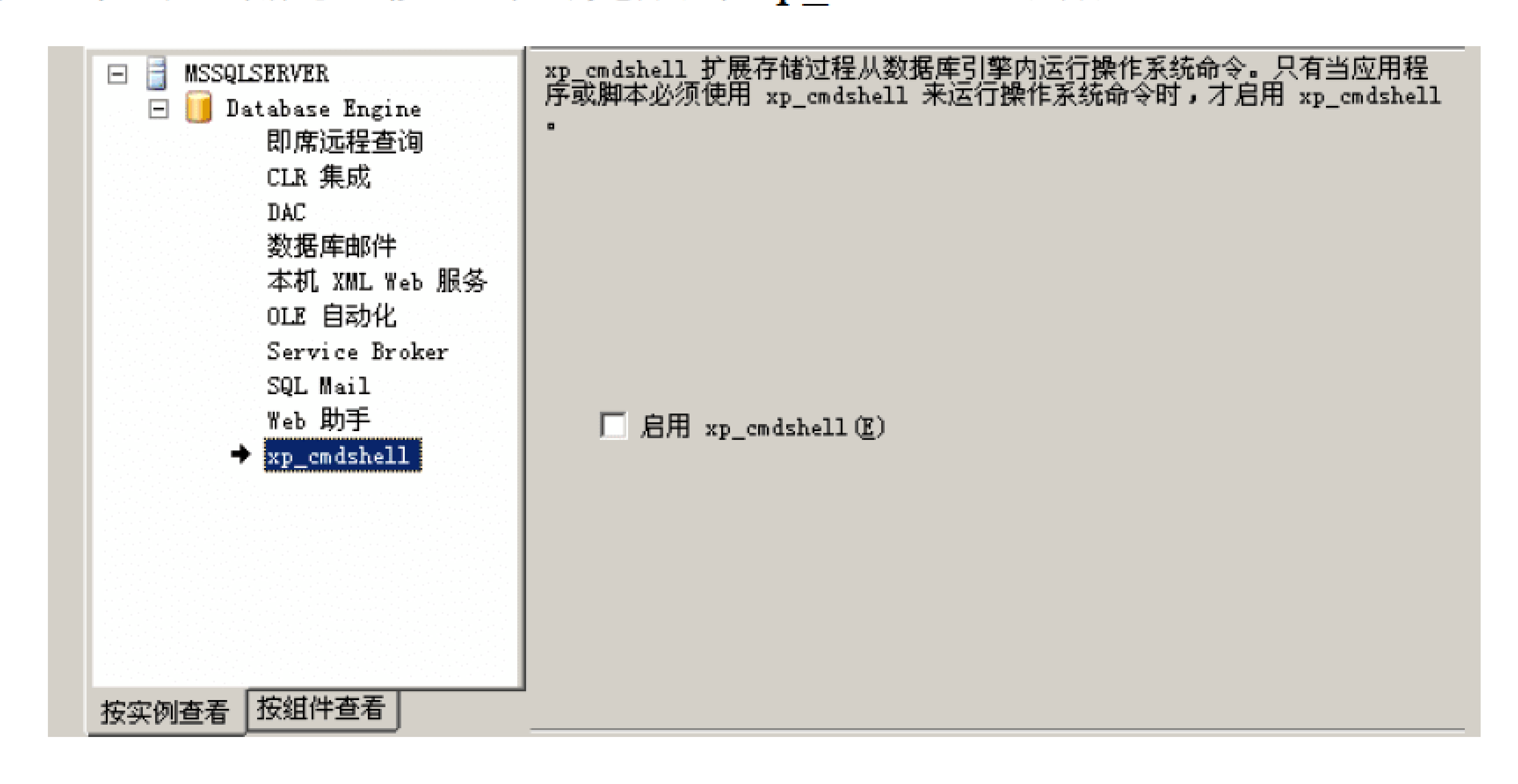


图 5.4　配置 xp_cmdshell

5.3　登录名管理

登录名是用户登录 SQL Server 的重要标识，若要登录到 SQL Server 数据库系统，主体必须具有有效的 SQL Server 登录名。在身份验证过程中会使用此登录名，以验证是否允许主体连接到该 SQL Server 实例。目前比较常用的管理 SQL Server 登录用户的方式有两种：使用 T-SQL 和使用 SSMS 可视化界面。本节将使用这两种方式讲解登录名的管理。

5.3.1　使用 T-SQL 创建登录名

SQL Server 提供了 CREATE LOGIN 命令用于创建登录名。该命令的语法格式如代码 5.2 所示。

代码 5.2　CREATE LOGIN 的语法

```
CREATE LOGIN loginName { WITH <option list1> | FROM <sources> }
<option list1> ::=
   PASSWORD = { 'password' | hashed_password HASHED } [ MUST_CHANGE ]
   [ , <option list2> [ ,... ] ]
<option_list2> ::=
   SID = sid
   | DEFAULT_DATABASE = database
```

```
    | DEFAULT LANGUAGE = language
    | CHECK EXPIRATION = { ON | OFF}
    | CHECK POLICY = { ON | OFF}
    | CREDENTIAL = credential name
<sources> ::=
   WINDOWS [ WITH <windows options> [ ,... ] ]
    | CERTIFICATE certname
    | ASYMMETRIC KEY asym key name
<windows options> ::=
   DEFAULT DATABASE = database
    | DEFAULT_LANGUAGE = language
```

其中比较重要的几个参数说明如下。

- loginName：指定创建的登录名。有 4 种类型的登录名：SQL Server 登录名、Windows 登录名、证书映射登录名和非对称密钥映射登录名。如果从 Windows 域账户映射 loginName，则 loginName 必须用方括号（[ ]）括起来。
- PASSWORD = 'password'：仅适用于 SQL Server 登录名，指定正在创建的登录名的密码。
- MUST_CHANGE：仅适用于 SQL Server 登录名。如果包括此选项，则 SQL Server 将在首次使用新登录名时提示用户输入新密码。
- DEFAULT_DATABASE = database：指定将指派给登录名的默认数据库。如果未包括此选项，则默认数据库将设置为 master。
- DEFAULT_LANGUAGE = language：指定将指派给登录名的默认语言。如果未包括此选项，则默认语言将设置为服务器的当前默认语言。即使将来服务器的默认语言发生更改，登录名的默认语言也仍保持不变。
- CHECK_EXPIRATION = { ON | OFF }：仅适用于 SQL Server 登录名，指定是否对此登录账户强制实施密码过期策略。默认值为 OFF。
- CHECK_POLICY = { ON | OFF }：仅适用于 SQL Server 登录名。指定应对此登录名强制实施运行 SQL Server 计算机的 Windows 密码策略。默认值为 ON。

注意：*只有在 Windows Server 2003 及更高版本上才能执行 CHECK_EXPIRATION 和 CHECK_POLICY。*

例如要创建一个登录名 testuser1，创建时的密码为 123456，由于该密码可能不符合 Windows 的密码策略，所以在该账户上不使用 Windows 密码策略。创建登录名的 SQL 脚本如代码 5.3 所示。

代码 5.3　使用 CREATE LOGIN 创建登录名

```
CREATE LOGIN testuser1
WITH PASSWORD='password1',
CHECK_POLICY =OFF --不启用 Windows 密码策略
```

如果要将当前 Windows 账户中的用户 SQLAdmin 添加到 SQL Server 登录用户中，则对应的 SQL 脚本为：

```
CREATE LOGIN [MS-ZY\SQLAdmin]
FROM WINDOWS --基于 Windows 认证
```

除了 CREATE LOGIN 命令外，SQL Server 还提供了 sp_addlogin 系统存储过程用于添加登录名，不过不推荐使用这种方式，SQL Server 可能在以后的版本中删除该存储过程。其使用语法如代码 5.4 所示。

代码 5.4　sp_addlogin 的语法

```
sp_addlogin [ @loginame = ] 'login'
    [ , [ @passwd = ] 'password' ]
    [ , [ @defdb = ] 'database' ]
    [ , [ @deflanguage = ] 'language' ]
    [ , [ @sid = ] sid ]
    [ , [ @encryptopt= ] 'encryption_option' ]
```

该存储过程中只有@loginame 参数是必须的，其他参数都提供了默认值，但是一般情况下需要指定@loginname 和@passwd 这两个参数。每个参数的含义如下所述。

- [@loginame=]'login'：登录的名称。login 的数据类型为 sysname，无默认值。
- [@passwd=]'password'：登录的密码。password 的数据类型为 sysname，默认值为 NULL。
- [@defdb=]'database'：登录的默认数据库（在登录后登录首先连接到该数据库）。database 的数据类型为 sysname，默认值为 master。
- [@deflanguage=]'language'：登录的默认语言。language 的数据类型为 sysname，默认值为 NULL。如果未指定 language，则新登录的默认 language 将设置为服务器的当前默认语言。
- [@sid=]'sid'：安全标识号(SID)。sid 的数据类型为 varbinary(16)，默认值为 NULL。如果 sid 为 NULL，则系统将为新登录生成 SID。不管是否使用 varbinary 数据类型，NULL 以外的值的长度都必须正好是 16 个字节，并且一定不能已经存在。
- [@encryptopt=]'encryption_option'：指定是以明文形式，还是以明文密码的哈希运算结果来传递密码。如果传入明文密码，将对它进行哈希运算，哈希值将存储起来。

说明：如果没有指定@passwd 参数，那么建立的用户将是空密码，这是很危险的，最好在建立用户时就将密码指定，而且最好使用强密码。

例如要建立一个用户 testuser1，该用户的密码是 password1，则只需要运行以下脚本。

```
EXEC sp_addlogin testuser1,'password1'
```

说明：sp_addlogin 的实质还是通过调用 CREATE LOGIN 命令来创建登录名，该存储过程只是对 CREATE LOGIN 命令的简化封装，读者可以运行 sp_helptext sp_addlogin 命令来查看该存储过程的定义。

5.3.2　使用 SSMS 创建登录名

在 SSMS 中创建登录名的操作步骤如下所述。

（1）使用对服务器拥有 ALTER ANY LOGIN 或 ALTER LOGIN 权限的用户（如 sa）

登录进 SQL Server。

（2）在对象资源管理器中展开“安全性”节点下的“登录名”节点并右击，在弹出的快捷菜单中选择“新建登录名”选项，系统将弹出“登录名-新建”对话框，如图 5.5 所示。

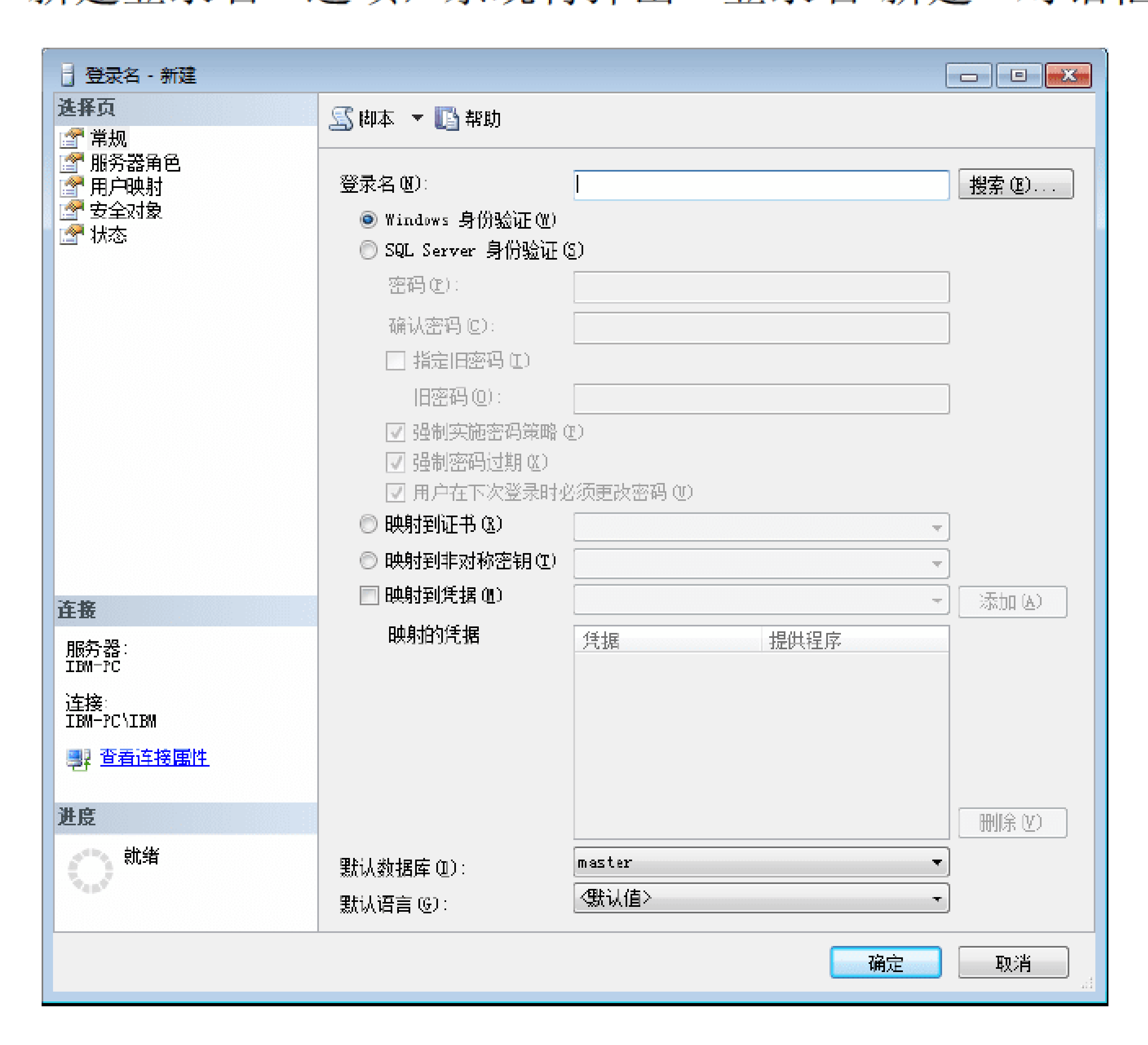


图 5.5　“登录名-新建”对话框

（3）如果要创建的登录名是 Windows 用户，则单击该对话框右侧的“搜索”按钮查找并选择 Windows 用户；如果是创建 SQL Server 登录名，则直接输入登录名（例如创建 testuser2）并选中“SQL Server 身份认证”单选按钮。

（4）对于 SQL Server 身份认证，需要输入密码和确认密码，两次输入的密码必须相同。

（5）对于 SQL Server 身份认证，若要实施密码策略，则选中“强制实施密码策略”复选框。在没有选中“强制密码过期”复选框的情况下是无法选中“用户在下次登录时必须更改密码”复选框的，如图 5.6 所示。

（6）选择默认数据库和默认语言，一般情况下就使用默认设置。

（7）单击“确定”按钮，如果启用了密码策略并且当前密码符合 Windows 的密码策略则创建用户成功。创建后的用户将出现在对象资源管理器中，如图 5.7 所示。

强制实施密码策略(F)
强制密码过期(X)
用户在下次登录时必须更改密码(U)

图 5.6　密码策略选项

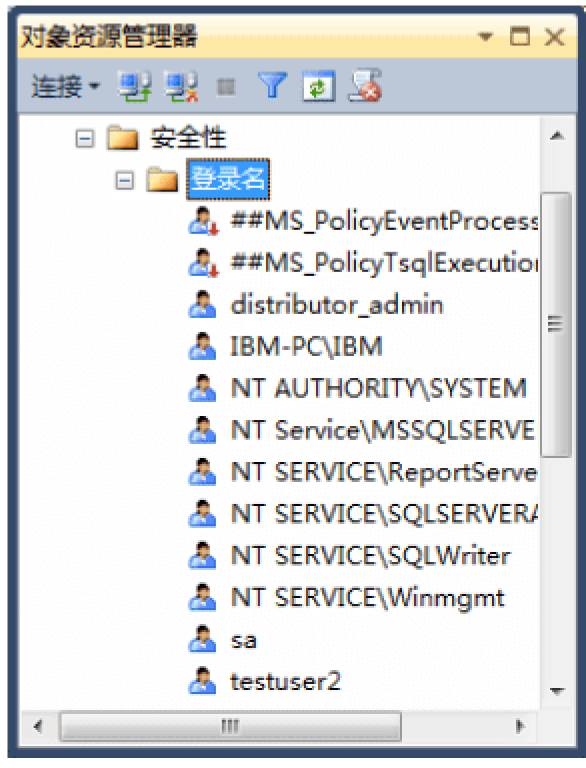


图 5.7　在对象资源管理器中查看登录名

5.3.3　使用 T-SQL 修改登录名

在创建好用户后，如希望对用户的密码进行更改，则需要使用 ALTER LOGIN 命令。该命令的语法如代码 5.5 所示。

代码 5.5　ALTER LOGIN 的语法

```
ALTER LOGIN login name
    {
    <status option>
    | WITH <set option> [ ,... ]
    | <cryptographic_credential_option>
    }
<status_option> ::=
        ENABLE | DISABLE
<set option> ::=
    PASSWORD = 'password' | hashed_password HASHED
    [
      OLD PASSWORD = 'oldpassword'
      | <password_option> [<password_option> ]
    ]
    | DEFAULT DATABASE = database
    | DEFAULT LANGUAGE = language
    | NAME = login name
    | CHECK_POLICY = { ON | OFF }
    | CHECK EXPIRATION = { ON | OFF }
    | CREDENTIAL = credential name
    | NO CREDENTIAL
<password option> ::=
    MUST_CHANGE | UNLOCK
<cryptographic credentials option> ::=
         ADD CREDENTIAL credential name
          | DROP CREDENTIAL credential_name
```

其中重要的参数介绍如下。

- ❑ login_name：指定正在更改的 SQL Server 登录的名称。
- ❑ ENABLE | DISABLE：启用或禁用此登录。
- ❑ PASSWORD = 'password'：仅适用于 SQL Server 登录账户。指定正在更改的登录的密码。密码是区分大小写的。
- ❑ OLD_PASSWORD = 'oldpassword'：仅适用于 SQL Server 登录账户，要指定当前密码。 密码是区分大小写的。
- ❑ MUST_CHANGE：仅适用于 SQL Server 登录账户。如果包括此选项，则 SQL Server 将在首次使用已更改的登录时提示输入更新的密码。
- ❑ DEFAULT_DATABASE = database：指定登录后的默认数据库。
- ❑ DEFAULT_LANGUAGE = language：指定登录后的默认语言。
- ❑ NAME = login_name：正在重命名的登录的新名称。 如果是 Windows 登录，则与

新名称对应的 Windows 主体的 SID，必须匹配与 SQL Server 中登录相关联的 SID。SQL Server 登录的新名称不能包含反斜杠字符（\）。

- CHECK_EXPIRATION = { ON | OFF }：仅适用于 SQL Server 登录账户。指定是否对此登录账户强制实施密码过期策略。默认值为 OFF。
- CHECK_POLICY = { ON | OFF }：仅适用于 SQL Server 登录账户。指定应对此登录名强制实施运行 SQL Server 的计算机的 Windows 密码策略。默认值为 ON。

使用 ALTER LOGIN 需要具有 ALTER ANY LOGIN 的权限。如果正在更改的登录名是 sysadmin 固定服务器角色的成员，或 CONTROL SERVER 权限的被授权者，则进行以下更改时还需要 CONTROL SERVER 权限：

- 在不提供旧密码的情况下重置密码。
- 启用 MUST_CHANGE、CHECK_POLICY 或 CHECK_EXPIRATION。
- 更改登录名。
- 启用或禁用登录。
- 将登录映射到其他凭据。

主体可更改用于自身登录的密码、默认语言，以及默认数据库。

例如，使用 sa 账户来修改前面创建的用户 testuser1 的密码为 abcdefg，则使用 ALTER LOGIN 的脚本为：

```
ALTER LOGIN testuser1
WITH PASSWORD='abcdefg'
```

以上代码的执行需要当前用户具有 ALTER ANY LOGIN 的权限，如果是使用刚创建的用户 testuser1 登录并执行以上 SQL 脚本，系统将抛出异常：

```
消息 15151，级别 16，状态 1，第 1 行
无法对 登录名 'testuser1' 执行 更改，因为它不存在，或者您没有所需的权限。
```

在没有 ALTER ANY LOGIN 权限的情况下，用户必须使用 OLD_PASSWORD 指定原密码，在原密码正确的情况下才能修改当前用户的密码。例如使用 testuser1 连接数据库，该用户的原密码是 password1，如果将该用户的密码修改为 abcdefg，则对应的 SQL 脚本如代码 5.6 所示。

代码 5.6　ALTER LOGIN 修改密码

```
ALTER LOGIN testuser1
WITH PASSWORD='abcdefg'
OLD_PASSWORD ='password1' --必须指定原密码
```

若希望修改登录名 testuser1 为 testuser11 或者禁用某登录名也是使用 ALTER LOGIN 命令。具体 SQL 脚本如代码 5.7 所示。

代码 5.7　修改、禁用登录名

```
ALTER LOGIN testuser1 --修改登录名
WITH NAME = testuser11
GO
```

```
ALTER LOGIN testuser11 DISABLE --禁用登录名
```

同创建登录名中可以使用 sp_addlogin 一样，系统还提供了 sp_password 存储过程用于修改密码，不过这种修改密码的方法已经过时，在将来的 SQL Server 系统中将会删除该功能。sp_password 的语法如代码 5.8 所示。

代码 5.8　sp_password 的语法

```
sp password [ [ @old = ] 'old password' , ]
    { [ @new =] 'new password' }
    [ , [ @loginame = ] 'login' ]
```

其中的@loginame 如果未指定，则表示修改当前用户的密码。该存储过程使用所需权限与 ALTER LOGIN 相同，若具有 ALTER ANY LOGIN 的权限可以不给出原密码。

说明： *sp_password 的实质还是调用了 ALTER LOGIN，该存储过程实际上就是对 ALTER LOGIN 命令修改密码功能的封装，使用 sp_helptext sp_password 命令可以看到该存储过程的定义，读者若有兴趣可以运行该命令查看其内容。*

5.3.4　使用 SSMS 修改登录名

使用 SSMS 修改前面创建的登录名 testuser2，其操作步骤如下所述。

（1）使用具有 ALTER ANY LOGIN 权限的账户登录系统。

（2）在对象资源管理器中展开“安全性”节点下的“登录名”节点。在“登录名”节点下找到前面创建的 testuser2 登录名。

（3）右击 testuser2 登录名，在弹出的快捷菜单中选择“属性”选项，或者直接双击 testuser2 登录名，系统将弹出“登录属性”对话框，如图 5.8 所示。

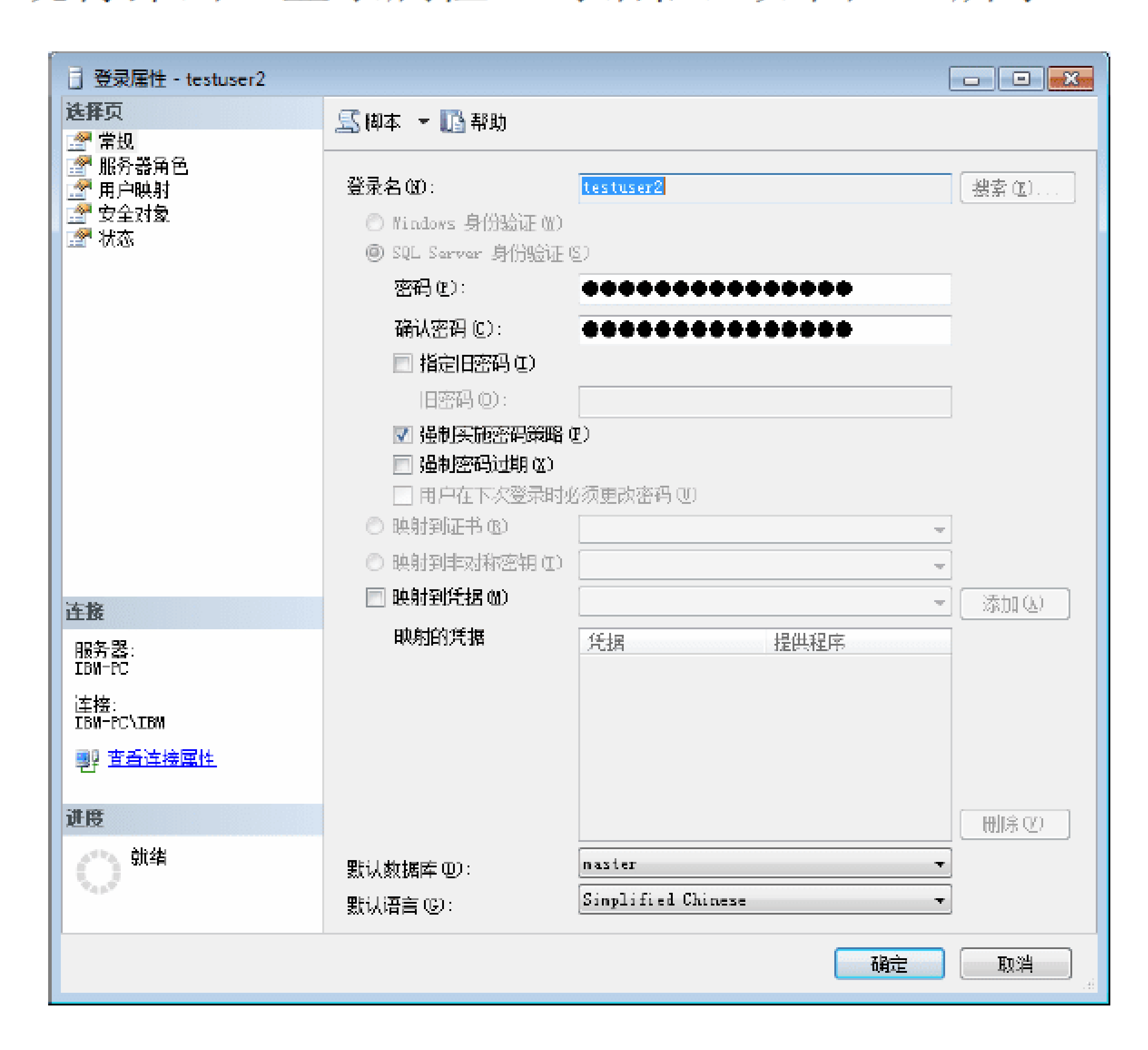


图 5.8　“登录属性”对话框

（4）若要重新设置密码，在密码文本框和确认密码文本框中可以输入密码。

注意：“登录属性”对话框中，密码文本框里*的个数并不是密码的长度，不论原密码有几位，这里都是用十几个*来表示。

（5）在“强制实施密码策略”文本框中可以修改该用户的密码策略和强制密码过期，但是无法选中“用户在下次登录时必须更改密码”复选框。

（6）最下面可以更改默认数据库和默认语言。

（7）若要修改登录名的状态，禁用该登录名或重新启用该登录名，需要单击左侧的“状态”链接，系统进入状态选项卡，如图 5.9 所示。

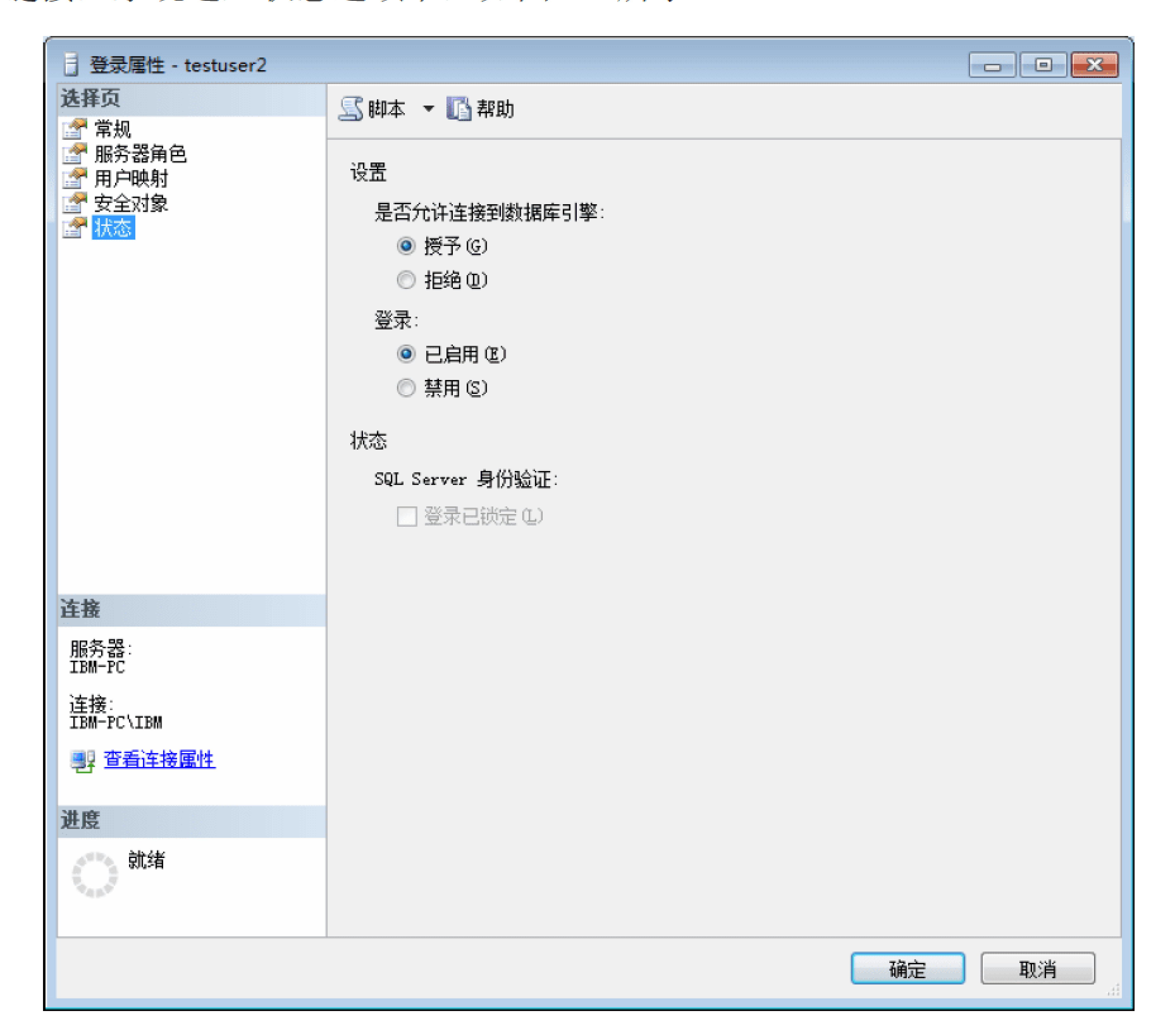


图 5.9　登录属性的状态选项卡

（8）在状态窗口可以选择“已启用”单选按钮或“禁用”单选按钮来启用或禁用登录名。

（9）单击“确定”按钮完成对登录名的修改。

如果需要对登录名进行重命名，只需要选中登录名使用快捷键 F2 或者右击某登录名，在弹出的快捷菜单中选中“重命名”选项，便可重新命名登录名。

5.3.5　删除登录名

使用 T-SQL 删除登录非常简单，SQL Server 提供了 DROP LOGIN 命令用于删除登录名，其语法为：

```
DROP LOGIN login_name
```

其不能删除正在登录的登录名，要删除登录名需要对服务器具有 ALTER ANY LOGIN

权限。例如要删除前面创建并重命名的登录名 testuser11，则运行脚本：

```
DROP LOGIN testuser11
```

使用 SSMS 删除登录名的方式与删除其他数据库对象（如表、存储过程和视图等）一样。在对象资源管理器中选中要删除的登录名，使用快捷键 Delete，系统将弹出删除对象窗口，单击“确定”按钮即可删除登录名。

5.4　用 户 管 理

在 5.3 节创建的登录名提供了登录权限，在登录后如果要让该用户具有数据库的访问权，那么要给该用户授予数据库访问权。授予数据库访问权是通过把一个用户（User）添加到需要访问的数据库的 Users 成员中实现的。本节将主要讲解对用户的管理操作。

5.4.1　使用 T-SQL 创建用户

在 5.3 节创建登录名时，如果使用非系统数据库，例如 AdventureWorks 2012 作为默认数据库，则使用该登录名将无法登录 SQL Server，SQL Server 将抛出异常“无法打开用户默认数据库，登录失败”，这是由于在 AdventureWorks2012 数据库中没有该登录名对应的用户。只有在登录名对应了数据库中的用户后该登录名才可以访问对应的数据库。在 T-SQL 下通过 CREATE USER 命令创建用户，CREATE USER 的语法如代码 5.9 所示。

代码 5.9　CREATE USER 的语法

```
CREATE USER user name    [ { { FOR | FROM }
      {
        LOGIN login name
        | CERTIFICATE cert name
        | ASYMMETRIC KEY asym key name
      }
      | WITHOUT LOGIN
    ]
    [ WITH DEFAULT_SCHEMA = schema_name ]
```

运行 CREATE USER 命令需要对数据库具有 ALTER ANY USER 权限。

如果已忽略 FOR LOGIN，则新的数据库用户将被映射到同名的 SQL Server 登录名。如果未定义 DEFAULT_SCHEMA，则数据库用户将使用 dbo 作为默认架构。

其中，user_name 指定在此数据库中用于识别该用户的名称。LOGIN login_name 指定要创建数据库用户的 SQL Server 登录名。login_name 必须是服务器中有效的登录名。当此 SQL Server 登录名进入数据库时，它将获取正在创建的数据库用户的名称和 ID。

注意：不能使用 CREATE USER 创建 guest 用户，因为每个数据库中均已存在 guest 用户。

例如，要创建登录名 testuser1 并在 AdventureWorks2012 数据库上创建同名的用户，则对应的 SQL 脚本如代码 5.10 所示。

代码 5.10　创建登录名和同名用户

```
CREATE LOGIN testuser1 --创建登录名
   WITH PASSWORD = 'abcdefg'; --指定密码
GO
USE AdventureWorks2012;
CREATE USER testuser1; --创建用户
GO
```

另外，若要创建的用户与登录名不相同，则必须要指定用户对应的登录名。例如创建登录名 testuser2，在数据库 AdventureWorks2012 中创建用户 test2 的 SQL 脚本如代码 5.11 所示。

代码 5.11　创建不同的登录名和用户

```
CREATE LOGIN testuser2
   WITH PASSWORD = 'abcdefg';
GO
USE AdventureWorks2012;
CREATE USER test2
FOR LOGIN testuser2 --必须指定对应的登录名
GO
```

注意：登录名和具体数据库中的用户是一对一的关系，也就是说，一个登录名在一个数据库中最多只能对应一个用户，而一个用户也只对应一个登录名。但是对应整个 SQL Server 实例来说，登录名与用户是一对多的关系，因为一个登录名可以在每一个数据库中创建不同的用户。

在未创建用户时运行 USE AdventureWorks2012 命令系统将抛出异常：

```
服务器主体 "testuser1" 无法在当前安全上下文下访问数据库 "AdventureWorks2012"。
```

在创建了用户与登录名的对应后，用户就可以正确运行 USE AdventureWorks2012 了。

技巧：若要查看当前连接在数据库中对应的用户，可以使用 CURRENT_USER()函数或者使用 USER_NAME()函数，运行 SELECT CURRENT_USER 即可。

5.4.2　使用 SSMS 创建用户

在 SSMS 的可视化界面下创建用户的主要操作如下所述。

（1）使用对 AdventureWorks2012 数据库具有 ALTER ANY USER 权限的用户，如 sa 登录进 SQL Server。

（2）在对象资源管理器中展开 AdventureWorks2012 数据库“安全性”节点“用户”下的节点。

（3）右击“用户”节点，在弹出的快捷菜单中选择“新建用户”选项，系统将弹出“数据库用户-新建”对话框，如图 5.10 所示。

（4）在“用户名”文本框中输入需要新建的用户名。

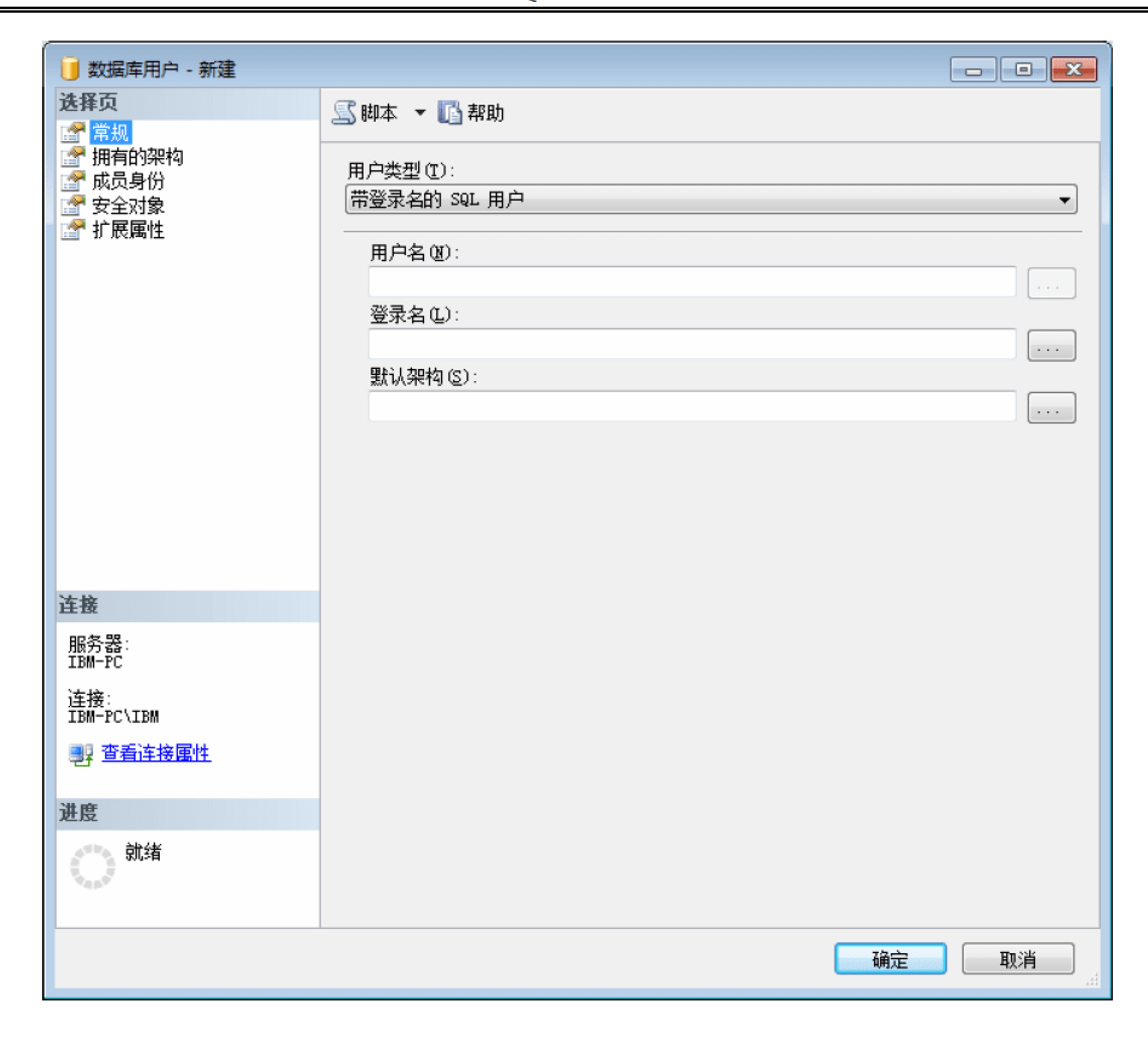


图 5.10　“数据库用户-新建”对话框

（5）在“登录名”文本框中输入登录名，也可以使用“登录名”文本框右侧的按钮来帮助选择登录名。

（6）“默认架构”文本框可以不填，系统将自动默认为 dbo。

（7）此用户拥有的架构和数据库角色成员身份两个选项暂不选择，这两个属性在接下来的章节中将进行讲解。

（8）单击“确定”按钮，用户就建立完成。建立的用户将在对象资源管理器中展示，如图 5.11 所示。

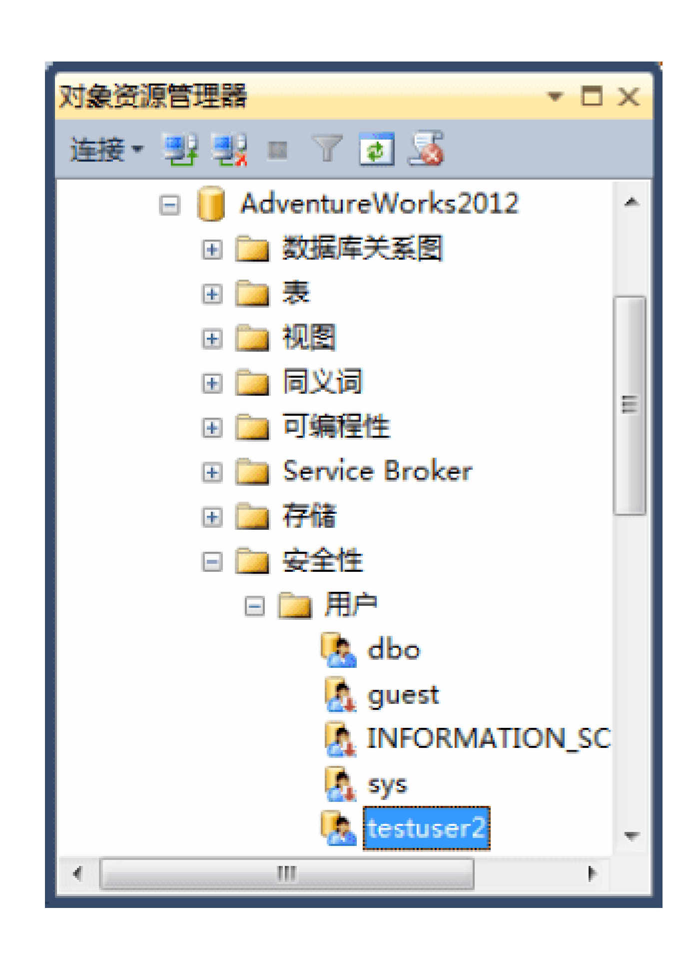


图 5.11　在对象资源管理器中查看用户

5.4.3　修改用户

修改用户使用 ALTER USER 命令。ALTER USER 的语法如代码 5.12 所示。

代码 5.12　ALTER USER 的语法

```
ALTER USER <user name>
    WITH <NAME = new user name
    | DEFAULT_SCHEMA = schema_name>
```

ALTER USER 的功能很简单，就是用于重命名数据库用户或更改它的默认架构。例如，要修改前面创建的用户 testuser1 为 test1 而默认的架构不变，则对应的 SQL 脚本如代码 5.13 所示。

代码 5.13　修改用户

```
USE AdventureWorks2012;
GO
ALTER USER testuser1 --修改用户的名字
    WITH NAME=test1
GO
```

使用 SSMS 修改用户与修改登录名类似，具体操作如下所述。

（1）使用对数据库具有 ALTER ANY USER 权限的用户登录。

（2）在对象资源管理器中双击需要修改的用户，系统将弹出“数据库用户”对话框，如图 5.12 所示。

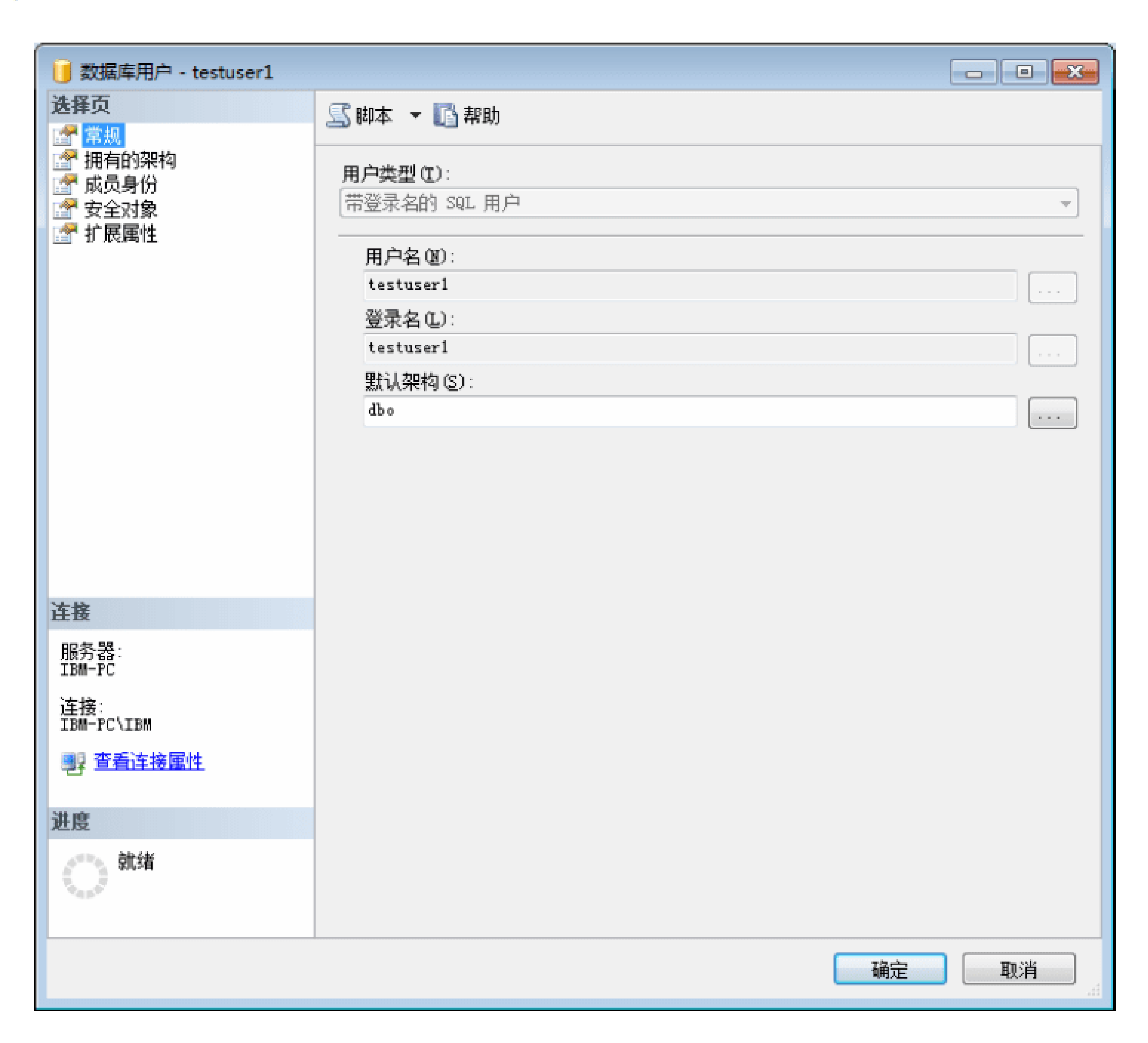


图 5.12　“数据库用户”对话框

（3）在该对话框中可以修改用户的默认架构，不能修改用户名。

（4）修改默认架构后单击“确定”按钮，默认架构修改完成。

注意：要修改用户名只能通过 T-SQL 进行修改，SSMS 不提供修改用户名的操作。

5.4.4　删除用户

若要删除用户，使用 DROP USER 命令即可。例如要删除前面创建的用户 test1，对应的脚本是：

```
DROP USER test1
```

在 SSMS 下删除用户的操作也和删除登录名相同。在对象资源管理器中找到需要删除的用户，然后使用快捷键 Delete，系统将弹出确认对话框，单击“确定”按钮即可完成对用户的删除。

5.5　架构管理

从 SQL Server 2005 起，架构行为已更改，架构不再等效于数据库用户；现在，每个架构都是独立于创建该架构的数据库用户而存在，架构与数据库用户是不同的命名空间。也就是说，架构只是对象的容器。架构与用户的分离方便了数据库的管理。本节将主要讲解架构的基础知识。

5.5.1　架构简介

架构（schema）是指包含表、视图、存储过程等数据库对象的容器。从包含关系上来讲，架构位于数据库内部，而数据库位于服务器内部。这些实体就像嵌套框放置在一起。服务器实例是最外面的框，而架构是最里面的框。

XML 架构集合、表、视图、过程、函数、聚合函数、约束、同义词、队列和统计信息等数据库对象都是必须位于架构内部的安全对象。特定架构中的每个安全对象都必须有唯一的名称，而不同的架构下可以有相同的数据库对象名称，例如，在一个数据库中可以同时存在 dbo.Department 表和 mis.Department 表。

架构中安全对象的完全指定名称包括此安全对象所在的架构名称。因此，架构也是命名空间。在默认情况下，系统的默认架构是 dbo。如果是访问默认架构中的对象则可以忽略架构名称，否则在访问表、视图等对象时需要指定架构名称，例如 mis.Customer 和 eip.vwDepartment 等。

注意：在 SQL Server 2000 和早期版本中，数据库可以包含一个名为“架构”的实体，但此实体实际上是数据库用户。在 SQL Server 2005、SQL Server 2008 和 SQL Server 2012 中，架构既是一个容器，又是一个命名空间。

从 SQL Server 2005 起将所有权与架构分离具有重要的意义：

- 架构的所有权和架构范围内的安全对象可以转移。
- 对象可以在架构之间移动。
- 单个架构可以包含由多个数据库用户拥有的对象。架构和用户之间是多对多的关系，一个用户可以拥有多个架构，一个架构也可以分给多个用户。
- 多个数据库用户可以共享单个默认架构。
- 与早期版本相比，对架构及架构中包含的安全对象权限的管理更加精细。
- 架构可以由任何数据库主体拥有，其中包括角色和应用程序角色。
- 可以删除数据库用户而不删除相应架构中的对象。删除用户并不会造成对架构和架构中对象的影响。

5.5.2　使用 T-SQL 创建架构

在 T-SQL 中创建架构需要使用 CREATE SCHEMA 命令，要运行该命令需要对数据库具有 CREATE SCHEMA 权限。CREATE SCHEMA 的语法结构如代码 5.14 所示。

代码 5.14　CREATE SCHEMA 的语法

```
CREATE SCHEMA schema_name_clause [ <schema_element> [ ...n ] ]
<schema_name_clause> ::=
    {
       schema_name
    | AUTHORIZATION owner_name
    | schema_name AUTHORIZATION owner_name
    }
<schema_element> ::=
    {
       table_definition | view_definition | grant_statement
       revoke_statement | deny_statement
    }
```

其中的参数说明如下所述。

- schema_name：在数据库内标识架构的名称。
- AUTHORIZATION owner_name：指定将拥有架构的数据库级主体的名称。此主体还可以拥有其他架构，并且可以不使用当前架构作为其默认架构。
- table_definition：指定在架构内创建表的 CREATE TABLE 语句。执行此语句的主体必须对当前数据库具有 CREATE TABLE 权限。
- view_definition：指定在架构内创建视图的 CREATE VIEW 语句。执行此语句的主体必须对当前数据库具有 CREATE VIEW 权限。
- grant_statement：指定可对除新架构外的任何安全对象授予权限的 GRANT 语句。
- revoke_statement：指定可对除新架构外的任何安全对象撤销权限的 REVOKE 语句。
- deny_statement：指定可对除新架构外的任何安全对象拒绝授予权限的 DENY 语句。

这里需要关注的是 schema_name 和 owner_name。新架构的拥有者可以是数据库用户、数据库角色或应用程序角色。在架构内创建的对象由架构所有者拥有。架构所包含对象的所有权可转让给任何数据库级别主体，但架构所有者始终保留对此架构内对象的

CONTROL 权限。

例如，要在 AdventureWorks2012 数据库中创建一个架构 t1，该架构的拥有者为前面创建的用户 test1。对应的 SQL 脚本如代码 5.15 所示。

代码 5.15　创建架构

```
USE AdventureWorks2012;
GO
CREATE SCHEMA t1
    AUTHORIZATION test1  --架构的拥有者为 test1 用户
```

注意： 在一个数据库中用户和架构是一对多的关系，也就是说可以为一个用户创建多个架构，但是一个架构只能指定一个拥有者。如果在 CREATE SCHEMA 中未指定拥有者，系统默认将 dbo 作为架构的拥有者。

5.5.3　使用 SSMS 创建架构

使用 SSMS 创建架构的主要操作步骤如下所述。

（1）使用对数据库具有 CREATE SCHEMA 权限的用户，例如 sa 登录系统。

（2）在对象资源管理器中展开 AdventureWorks2012 数据库的“安全性”节点下的“架构”节点。

（3）右击“架构”节点，在弹出的快捷菜单中选择“新建架构”选项，系统将弹出“架构-新建”对话框，如图 5.13 所示。

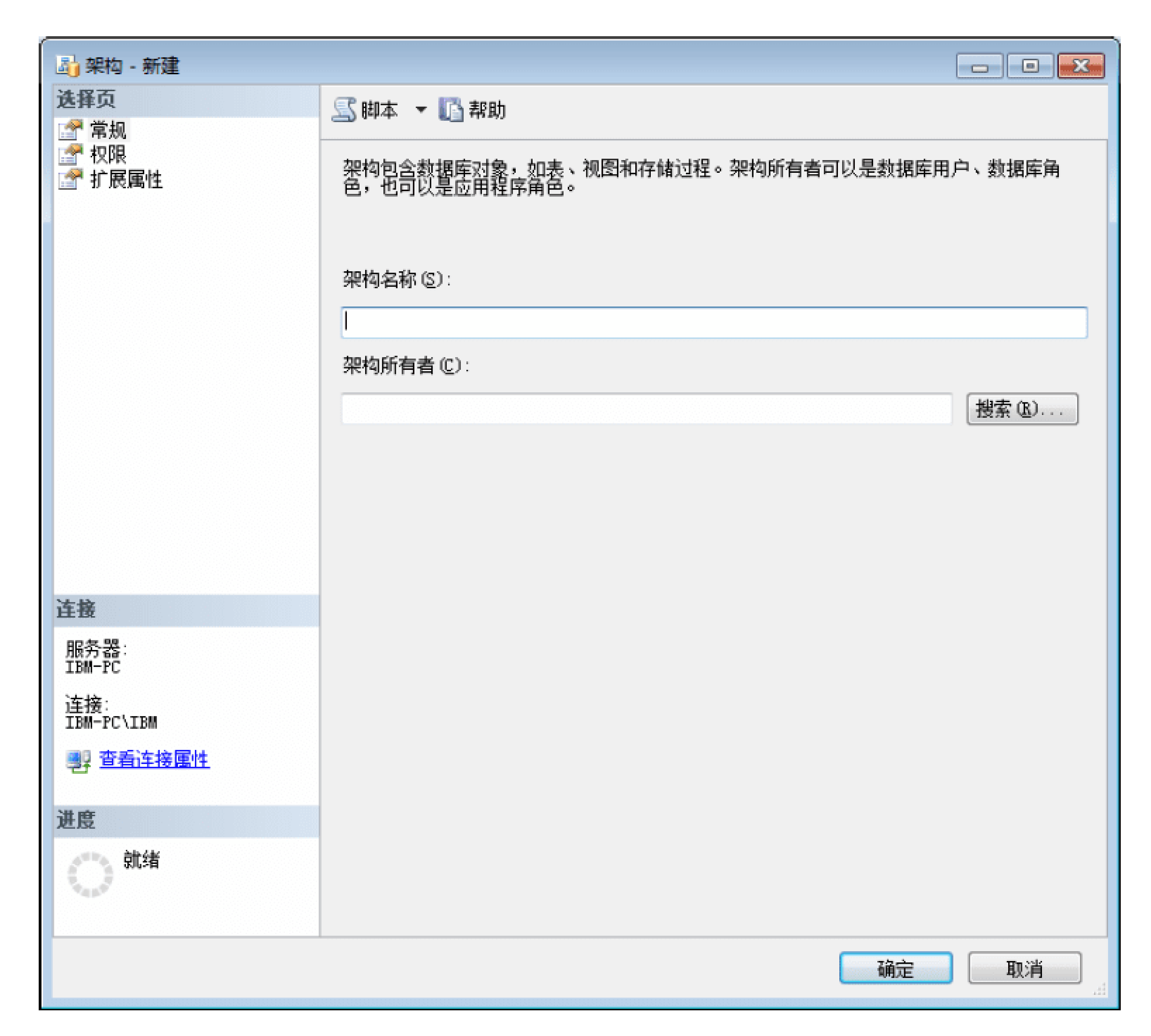


图 5.13　“架构-新建”对话框

（4）在“架构名称”文本框中输入架构的名称。

（5）在“架构所有者”文本框中输入架构的所有者用户，或者单击“搜索”按钮，系统将弹出“搜索角色和用户”对话框，如图 5.14 所示。

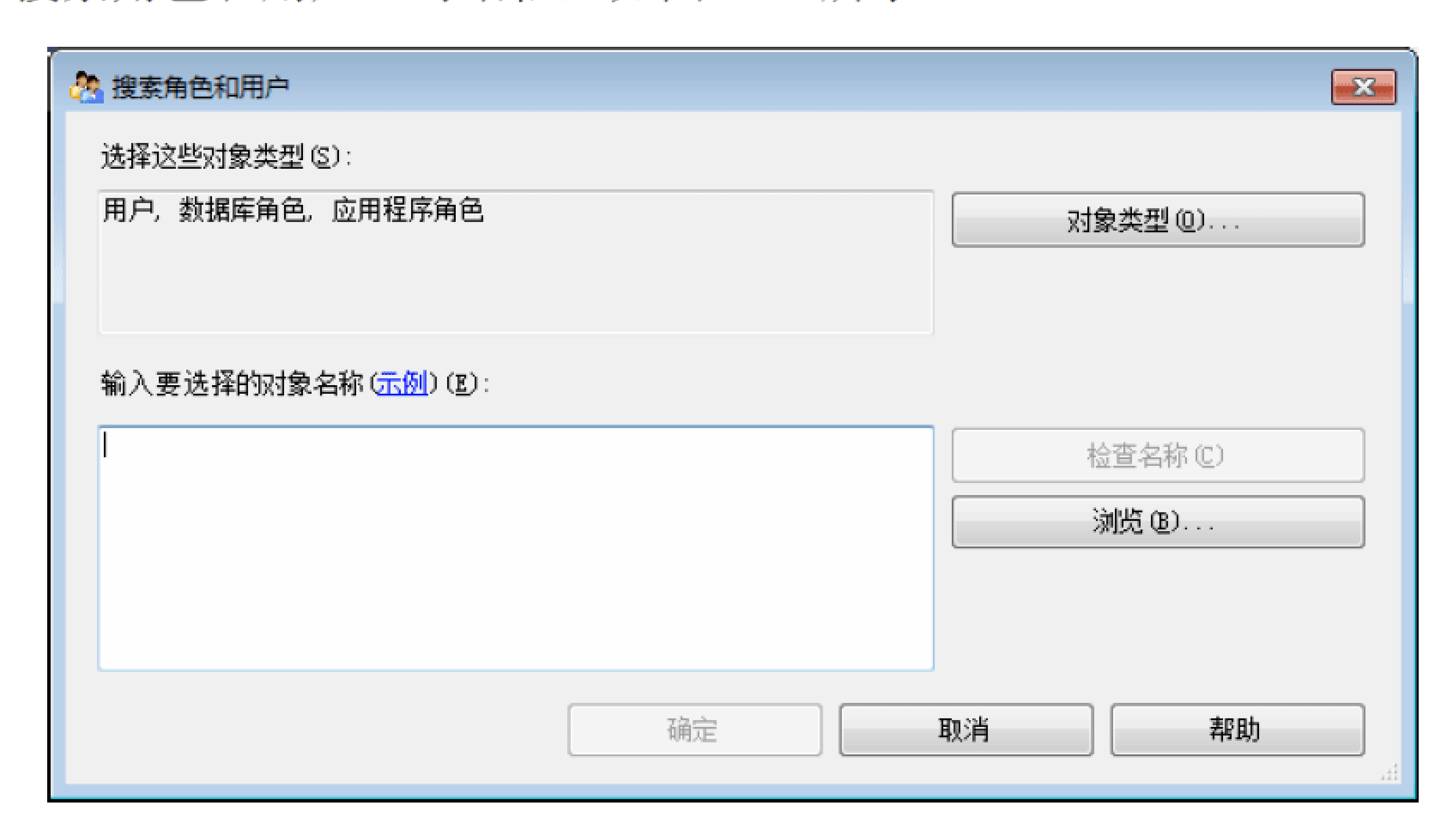


图 5.14　“搜索角色和用户”对话框

（6）单击“浏览”按钮系统将弹出“查找对象”对话框，其中列出了数据库中所有的角色和用户，如图 5.15 所示。

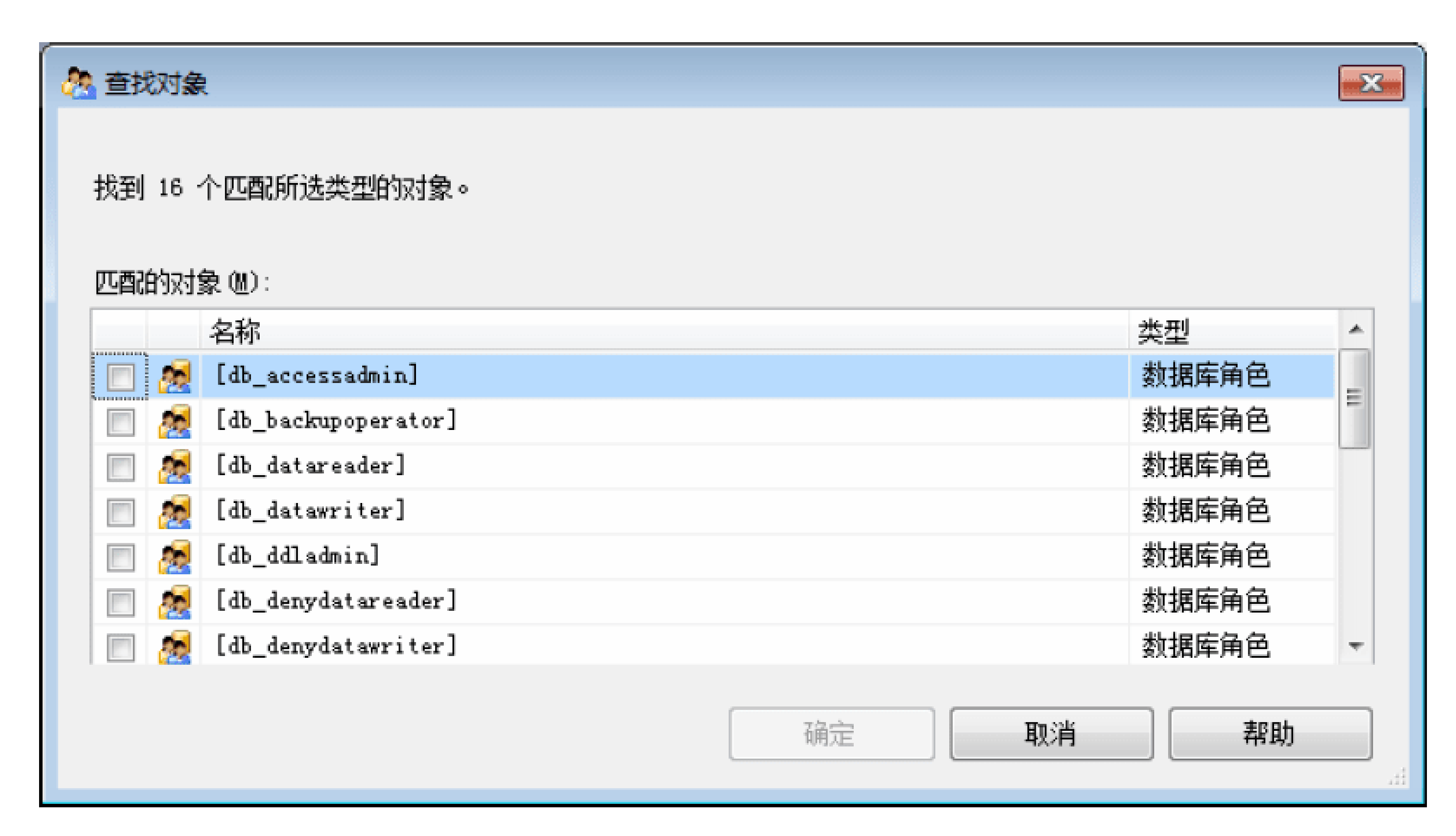


图 5.15　“查找对象”对话框

（7）在其中选中架构的拥有者对象并单击“确定”按钮，最后在“架构-新建”对话框中单击“确定”按钮完成架构的创建工作。

5.5.4　修改架构

若要修改架构需要使用 ALTER SCHEMA 命令，该命令的语法格式为：

```
ALTER SCHEMA schema_name TRANSFER securable_name
```

其中，schema_name 用于指定当前数据库中的架构名称，安全对象将移入其中。其数据类型不能为 SYS 或 INFORMATION_SCHEMA。securable_name 为要移入架构中的原架

构中所包含的安全对象的名称。

ALTER SCHEMA 仅可用于在同一数据库中的架构之间移动安全对象。若要更改或删除架构中的安全对象，可使用特定于该安全对象的 ALTER 或 DROP 语句。例如在 AdventureWorks2012 中创建了表 Student，该表所使用的架构是 dbo，现在需要将该表的架构修改为 t1，则对应的 SQL 脚本如代码 5.16 所示。

代码 5.16　使用 ALTER SCHEMA

```
USE [AdventureWorks2012]
GO
CREATE TABLE dbo.Student
(
    sID int IDENTITY PRIMARY KEY,
    sName nvarchar(10) NOT NULL
)
GO
ALTER SCHEMA t1 --修改架构
    TRANSFER dbo.Student
```

若要修改架构的所有者则需要使用 ALTER AUTHORIZATION 命令，该命令的语法格式如代码 5.17 所示。

代码 5.17　ALTER AUTHORIZATION 的语法

```
ALTER AUTHORIZATION
   ON [ <entity_type> :: ] entity_name
   TO { SCHEMA OWNER | principal_name }
<entity_type> ::=
    {
       Object | Type | XML Schema Collection | Fulltext Catalog | Schema
    | Assembly | Role | Message Type | Contract | Service
    | Remote Service Binding | Route | Symmetric Key | Endpoint
    | Certificate | Database
    }
```

ALTER AUTHORIZATION 用于更改安全对象的所有权。其中“<entity_type> ::”是更改其所有者实体的类。Object 是默认值，entity_name 是实体名。principal_name 为将拥有实体的主体名称。

ALTER AUTHORIZATION 可用于更改任何具有所有者实体的所有权。数据库包含实体的所有权，可以传递给任何数据库级的主体。服务器级实体的所有权只能传递给服务器级主体。例如要将架构 t1 的所有者修改为 test2 用户，则对应的 SQL 脚本为：

```
ALTER AUTHORIZATION ON SCHEMA::[t1] TO [test2]
```

5.5.5　删除架构

若要删除某架构，则需要使用 DROP SCHEMA 命令，该命令的语法很简单：

```
DROP SCHEMA schema_name
```

其中，schema_name 为架构在数据库中所使用的名称，要删除的架构不能包含任何对象。如果架构包含对象，则 DROP 语句将失败。

例如要删除前面用到的 t1 架构，则需要执行：

```
DROP SCHEMA t1
```

执行该命令时系统将抛出异常：

```
无法对 't1' 执行 drop schema，因为对象 'PK__Student__DDDED94E405A880E' 正引用它
```

这时需要将前面创建的 Student 表删除，或者使用 ALTER SCHEMA 命令将该表的架构修改为其他架构。

正确的删除 t1 架构的脚本如代码 5.18 所示。

代码 5.18　删除架构

```
USE [AdventureWorks2012]
GO
ALTER SCHEMA dbo
    TRANSFER t1.Student  --修改 Student 表的架构为 dbo
GO
DROP SCHEMA t1  --删除 t1 架构
```

在 SSMS 下删除架构也是和其他对象的删除一样使用 Delete 键即可，此处不再详述。

5.6　用户权限

简单地讲，用户权限就是规定用户可以做什么不可以做什么，在 5.5 节创建了登录用户后，本节将讲解用户权限的基础知识和权限的维护。

5.6.1　权限简介

SQL Server 2012 数据库引擎管理着可以通过权限保护实体的分层集合。这些实体称为“安全对象”。在安全对象中，从服务器实例和数据库，到更细的级别上的表、视图等对象都可以设置离散权限。SQL Server 通过验证主体是否已获得适当的权限，来控制主体对安全对象执行的操作。在 SQL Server 用户权限可以分为 3 类：

- 登录权限。
- 访问特定数据库的权限。
- 在数据库特定对象上执行特定行为的权限。

前面讲到的登录名就是用于登录权限，用户就是访问特定数据库的权限，本节讲的就是数据库特定对象的权限。如图 5.16 显示了 SQL Server 数据库引擎权限层次结构之间的关系。

在 5.1 节中已经介绍了按照容器从大到小来分，安全对象范围有服务器、数据库和架构。各个容器中又有各自的安全对象，对于每一个容器和每一个安全对象都可以控制用户

的访问权限。

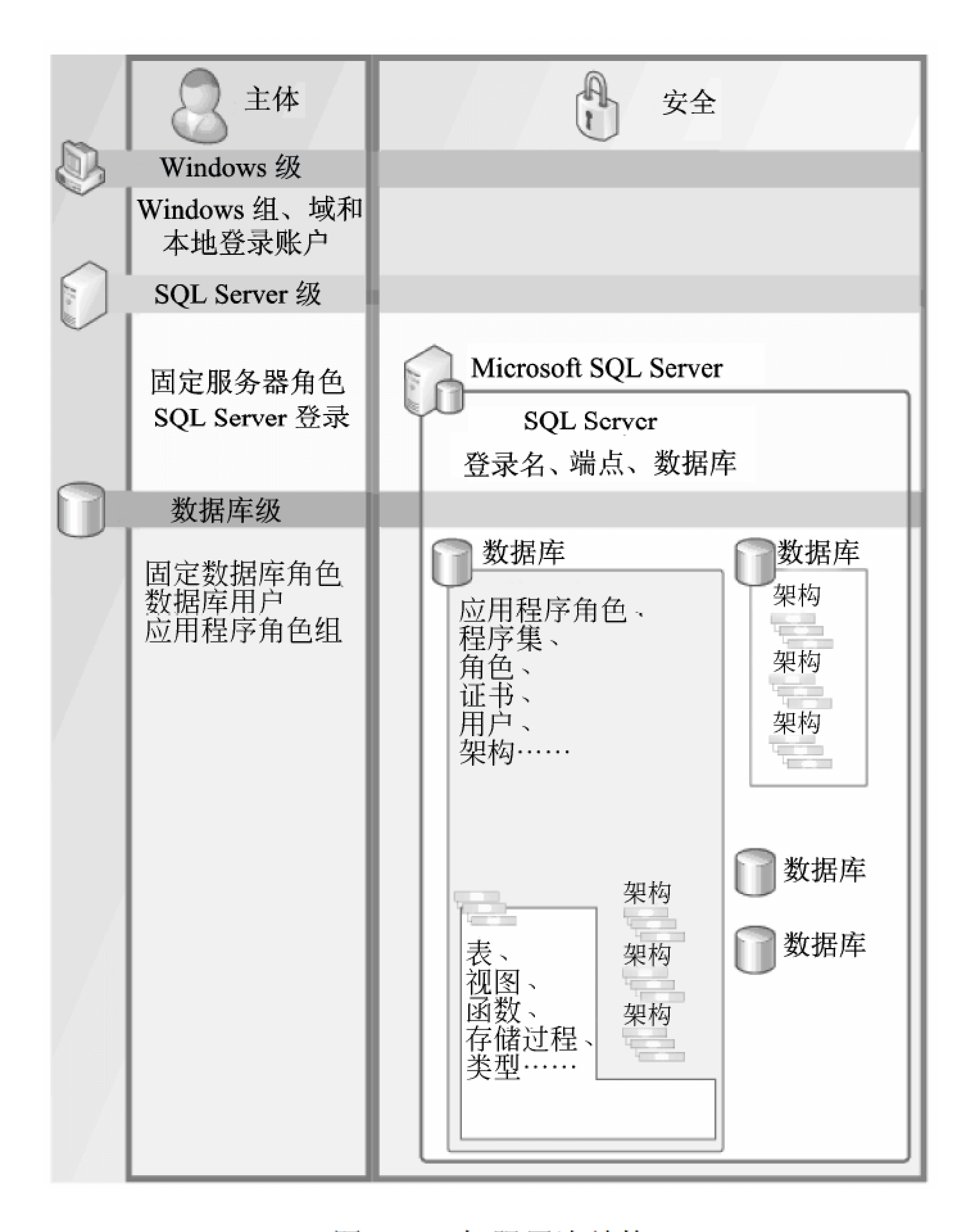


图 5.16　权限层次结构

通常使用 T-SQL 中的 GRANT、DENY 和 REVOKE 来操作权限，在设置权限时需要指定权限的名称，不同的权限使用不同的名称来表示。SQL Server 中对权限的名称有如下约定。

- CONTROL：表示被授权者授予类似所有权的功能。CONTROL 权限可以被看做是一个权限的集合，被授权者实际上对安全对象具有所定义的所有权限。SQL Server 安全模型是分层的，所以对某个数据库集合授予 CONTROL 权限，意味着对该集合范围内的所有安全对象具有 CONTROL 权限。例如，对某数据库架构具有 CONTROL 权限，隐含着对数据库架构下的所有表、视图、存储过程等，该架构下的所有对象具有的所有 CONTROL 权限。对一个表具有 CONTROL 权限意味着对该表有查询、修改、删除等该对象上的所有权限。
- ALTER：表示授予更改特定安全对象属性（所有权除外）的权限，ALTER 权限相对于 CONTROL 权限范围要小一些。如果对某个范围具有 ALTER 权限时，那么同时也具有了更改、创建或删除该范围内包含的任何安全对象的权限。例如，对表的 ALTER 权限包括在该表中创建、更改和删除的权限，但是并不具有 SELECT 权限，而如果为表指定的是 CONTROL 权限时，则具有 SELECT 权限。

- ALTER ANY <服务器安全对象>：其中的服务器安全对象可以是任何前面说到的服务器安全对象。授予创建、更改或删除服务器安全对象的各个实例的权限。例如，ALTER ANY LOGIN 将授予创建、更改或删除实例中的任何登录名的权限。
- ALTER ANY <数据库安全对象>：其中的数据库安全对象可以是数据库级别的任何安全对象。授予创建、更改或删除数据库安全对象的各个实例的权限。例如，ALTER ANY SCHEMA 将授予创建、更改或删除数据库中的所有架构的权限。
- TAKE OWNERSHIP：表示允许被授权者获取所授予的安全对象的所有权。
- IMPERSONATE <登录名>：表示允许被授权者模拟该登录名。
- IMPERSONATE <用户>：表示允许被授权者模拟该用户。
- CREATE <服务器安全对象>：授予被授权者创建服务器安全对象的权限。
- CREATE <数据库安全对象>：授予被授权者创建数据库安全对象的权限。
- CREATE <包含在架构中的安全对象>：授予创建包含在架构中的安全对象的权限。但是，若要在特定架构中创建安全对象，必须对该架构具有 ALTER 权限。
- VIEW DEFINITION：表示允许被授权者访问元数据。
- BACKUP 和 DUMP 是同义词。
- RESTORE 和 LOAD 是同义词。

如表 5.1 列出了主要的权限类别，以及可应用这些权限的安全对象的种类。

表 5.1　权限适用的安全对象

权　　限	适　用　于
SELECT	同义词
	表和列
	表值函数和列
	视图和列
VIEW CHANGE TRACKING	表
	架构
UPDATE	同义词
	表和列
	视图和列
REFERENCES	标量函数和聚合函数
	Service Broker 队列
	表和列
	表值函数和列
	视图和列
INSERT	同义词
	表和列
	视图和列
DELETE	同义词
	表和列
	视图和列
EXECUTE	过程
	标量函数和聚合函数
	同义词

续表

权　　限	适　用　于
RECEIVE	Service Broker队列
VIEW DEFINITION	过程
	Service Broker队列
	标量函数和聚合函数
	同义词
	表
	表值函数
	视图
ALTER	过程
	标量函数和聚合函数
	Service Broker队列
	表
	表值函数
	视图
TAKE OWNERSHIP	过程
	标量函数和聚合函数
	同义词
	表
	表值函数
	视图
CONTROL	过程
	标量函数和聚合函数
	Service Broker队列
	同义词
	表
	表值函数
	视图

5.6.2　使用 GRANT 分配权限

GRANT 用于给特定用户和角色授予了该对象指定的访问权限。GRANT 语句的语法如代码 5.19 所示。

代码 5.19　GRANT 的语法

```
GRANT { ALL [ PRIVILEGES ] }
      | permission [ ( column [ ,...n ] ) ] [ ,...n ]
      [ ON [ class :: ] securable ] TO principal [ ,...n ]
      [ WITH GRANT OPTION ] [ AS principal ]
```

其中，ALL 关键字表示希望给该类型的对象授予所有可用的权限。不推荐使用此选项，保留此选项仅用于向后兼容。授予 ALL 参数相当于授予以下权限：

- 如果安全对象为数据库，则 ALL 表示 BACKUP DATABASE、BACKUP LOG、CREATE DATABASE、CREATE DEFAULT、CREATE FUNCTION、CREATE

PROCEDURE、CREATE RULE、CREATE TABLE 和 CREATE VIEW 等权限。

- 如果安全对象为标量函数，则 ALL 表示 EXECUTE 和 REFERENCES。
- 如果安全对象为表值函数，则 ALL 表示 DELETE、INSERT、REFERENCES、SELECT 和 UPDATE。
- 如果安全对象是存储过程，则 ALL 表示 EXECUTE。
- 如果安全对象为表，则 ALL 表示 DELETE、INSERT、REFERENCES、SELECT 和 UPDATE。
- 如果安全对象为视图，则 ALL 表示 DELETE、INSERT、REFERENCES、SELECT 和 UPDATE。

PRIVILEGES 关键字是用于提供 ANSI-92 的兼容，并没有实际意义。ON 关键字显示了下一步出现的要授予权限的对象。在对表进行授权时可以指定影响的列清单并说明许可权，如果不说明则默认为所有列。TO 语句指定了想给哪个登录 ID 或角色名授权。WITH GRANT OPTION 允许正在授权的用户将同样的权限授予其他用户。AS 关键字是用于处理登录属于多个角色的问题。

在前面已经建立了登录名和用户从而实现了登录权限和数据库访问权限，但是当使用建立的登录名连接数据库后将无法看到具有访问权限的数据库中的表、视图和存储过程等安全对象，下面就使用 GRANT 命令将特定权限授予该登录名。

在数据库中创建登录名 testuser1，然后在 AdventureWorks2012 数据库中创建该登录名对应的用户 test1，若需要将表 Person.Address 的 SELECT 权限授予该用户，则对应的 SQL 脚本如代码 5.20 所示。

代码 5.20　使用 GRANT 授予表的 SELECT 权限

```
USE AdventureWorks2012;
GRANT SELECT ON Person.Address --授予 SELECT 权限
TO test1
```

这里需要注意的是，GRANT 授予权限给了用户 test1 而不是给登录名 testuser1。运行代码 5.20 后，使用 testuser1 登录将可以看到 AdventureWorks2012 数据库中有一个表 Person.Address，如图 5.17 所示。

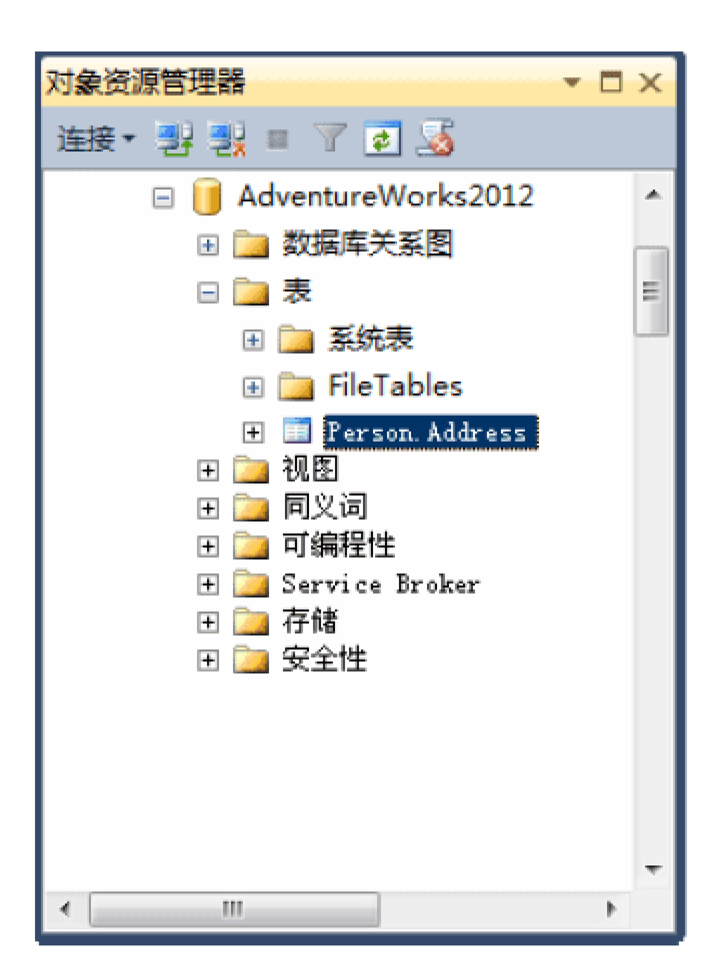


图 5.17　使用 testuser1 登录后看到的内容

同样地，如果要将存储过程 GetDeparment 的执行权限授予 test1 用户，则对应的 SQL 授权脚本如代码 5.21 所示。

代码 5.21　使用 GRANT 授予存储过程的执行权限

```
USE AdventureWorks2012;
GRANT EXECUTE ON dbo.GetDeparment --授予执行权限
TO test1
```

注意：若只是授予了 EXECUTE 权限给存储过程，那么该用户只有执行权限，而无法查看该存储过程的定义，在该用户看来存储过程相当于是被加密了的。

5.6.3　使用 DENY 显式拒绝访问对象

DENY 命令用于显式的拒绝用户访问指定的目标对象。SQL Server 中采用“拒绝大于一切”的权限管理机制，如果一个用户属于某个角色或拥有某个架构，而这些角色和架构中定义了对某数据库对象的访问权限，若该用户也被定义了拒绝访问该对象，则最终该用户将无法访问该对象，这就是“拒绝大于一切”的权限管理机制。DENY 命令的语法格式与 GRANT 的语法格式相似，其语法如代码 5.22 所示。

代码 5.22　DENY 的语法

```
DENY { ALL [ PRIVILEGES ] }
     | permission [ ( column [ ,...n ] ) ] [ ,...n ]
     [ ON [ class :: ] securable ] TO principal [ ,...n ]
     [ CASCADE] [ AS principal ]
```

其中：

- ALL 关键字用于表示拒绝该对象上可以应用的所有权限，如果不是 ALL 关键字，那么就要在被拒绝的对象上提供一个或多个特定许可权。
- PRIVILEGES 同样是为了提供兼容性而使用的。
- ON 关键字后跟要拒绝的对象。

以上关键字与 GRANT 命令中的意义相同，CASCADE 关键字与 GRANT 语句中的 WITH GRANT OPTION 对应，CASCADE 表示拒绝该用户已经在 WITH GRANT OPTION 规则下授予访问权的任何人。

同样以前面提到的 testuser1 为例，前面已经将 Person.Address 的 SELECT 权限授予了该登录名对应的用户 test1，现在要将整个 HumanResources 架构的 SELECT 权限授予该用户，但是不希望该用户查看 HumanResources.Employee 表，这里需要显式拒绝访问该表。具体执行脚本如代码 5.23 所示。

代码 5.23　使用 DENY 显式拒绝访问表

```
use [AdventureWorks2012]
GO
GRANT SELECT ON SCHEMA::HumanResources
TO test1 --查看该架构下的所有表
GO
```

```
--拒绝查看 HumanResources.Employee 表
DENY SELECT ON HumanResources.Employee
TO test1
```

5.6.4　使用 REVOKE 撤销权限

REVOKE 语句消除了以前执行 GRANT 或 DENY 语句的影响，把这条语句当做“撤销”语句。REVOKE 语句的语法与 GRANT 和 DENY 类似，如代码 5.24 所示。

代码 5.24　REVOKE 语句的语法格式

```
REVOKE [ GRANT OPTION FOR ]
      {
        [ ALL [ PRIVILEGES ] ]
        | permission [ ( column [ ,...n ] ) ] [ ,...n ]
      }
      [ ON [ class :: ] securable ]
      { TO | FROM } principal [ ,...n ]
      [ CASCADE] [ AS principal ]
```

同样地，ALL 关键字表示要撤销对象类型的所有权限，如果不使用 ALL，则必须指定要撤销该对象的一个或多个特定许可权。

- PRIVILEGES 仍然是维持兼容性的一个关键字。
- ON 关键字后显示了要撤销权限的对象。
- CASCADE 关键字与 GRANT 语句中的 WITH GRANT OPTION 对应，CASCADE 表示要撤销在 WITH GRANT OPTION 规则下授予用户的权限。
- AS 关键字说明了希望基于哪个角色执行该命令。

仍然以前面使用到的 testuser1 为例，前面已经对该登录名对应的用户 test1 授予了 HumanResources 架构和 Person.Address 表的 SELECT 权限，另外还拒绝了对 HumanResources.Employee 表的 SELECT 权限。这里若不希望再对 HumanResources 架构具有 SELECT 权限，则撤销该权限的脚本如代码 5.25 所示。

代码 5.25　使用 REVOKE 撤销权限

```
USE AdventureWorks2012
GO
REVOKE SELECT ON SCHEMA::HumanResources  --撤销对架构的 SELECT 权限
TO test1
```

注意：撤销了对 HumanResources 架构的 SELECT 权限，并不会同时撤销对 HumanResources.Employee 表的拒绝访问权限。虽然 HumanResources.Employee 表属于 HumanResources 架构，但是在 SQL Server 中将分别作为单独的数据库对象来对待。

5.6.5　语句执行权限

前面介绍的都是针对数据库对象的权限操作，但是数据库权限并不仅仅局限于数据库

对象，另外也包括不能立即连接到特定对象的某种 SQL 语句。SQL Server 提供了控制权限的许多语句，包括 CREATE DATABSE、CREATE DEFAULT、CREATE PROCEDURE、CREATE RULE、CREATE TABLE、CREATE VIEW、BACKUP DATABASE 和 BACKUP LOG。这些语句已经在前面的章节中已经做了介绍，这里就不再详述。对于这些语句，它们的权限控制其实和其他数据库对象的权限控制是类似的。

这里以创建数据库为例，要为 testuser1 登录名创建数据库的权限，那么首先该登录名必须要有对 master 数据库的访问权限。所以需要先在 master 中创建该登录名对应的用户，然后再为该用户授予 CREATE DATABASE 权限。具体 SQL 脚本如代码 5.26 所示。

代码 5.26　授予用户创建数据库权限

```
USE master;
GO
CREATE USER master1
FOR LOGIN testuser1
GO
GRANT CREATE DATABASE --授予创建数据库权限
TO master1
```

运行代码 5.26 后便可以使用 testuser1 登录然后创建数据库。默认情况下，该登录名将在创建的数据库中创建对应的用户 dbo，该用户对数据库内的对象具有完全的访问权限。

注意：只为用户授予了 CREATE DATABASE 权限后，该用户可以创建数据库也可以删除其创建的数据库，但是该用户不能修改其创建的数据库。用户必须要对数据库拥有 ALTER 权限才能使用 ALTER DATABASE 命令删除数据库。

同样，如果要拒绝用户的语句执行权限则使用 DENY 命令，如拒绝用户 master1 的创建数据库的权限，则对应的 SQL 脚本如代码 5.27 所示。

代码 5.27　拒绝创建数据库

```
USE master
GO
DENY CREATE DATABASE --拒绝创建数据库权限
TO master1
```

如果需要撤销对用户的语句执行权限的限制，则使用 REVOKE 命令。具体 SQL 脚本如代码 5.28 所示。

代码 5.28　撤销创建数据库权限

```
USE master
GO
REVOKE CREATE DATABASE --撤销创建数据库权限
TO master1
```

最后，若不再使用 testuser1 的创建数据库权限，也不再使用 master 数据库，那么可以执行：

```
DROP USER master1
```

删除该登录名在 master 数据库中的用户，从而禁止对 master 数据库的访问。

5.6.6　使用 SSMS 管理用户权限

SSMS 同样提供了大部分用户权限的管理界面，用户可以通过 SSMS 简单、快捷方便地设置用户的权限。假设新建一个登录名 testuser2，密码为 123，该登录名对 AdventureWorks2012 数据库下的 Person 架构下的表有查询权限，对 Person.AddressType 表有更改权限，那么要实现这样的配置需要以下几步操作。

（1）使用 sa 或者 Windows 账户登录 SSMS，在对象资源管理器中展开“安全性”节点下的“登录名”节点。

（2）右击“登录名”节点，在弹出的快捷菜单中选择“新建登录名”选项，系统弹出“登录名-新建”对话框。

（3）在“常规”选项的右边窗口中输入新建的登录名和密码等信息，如图 5.18 所示。

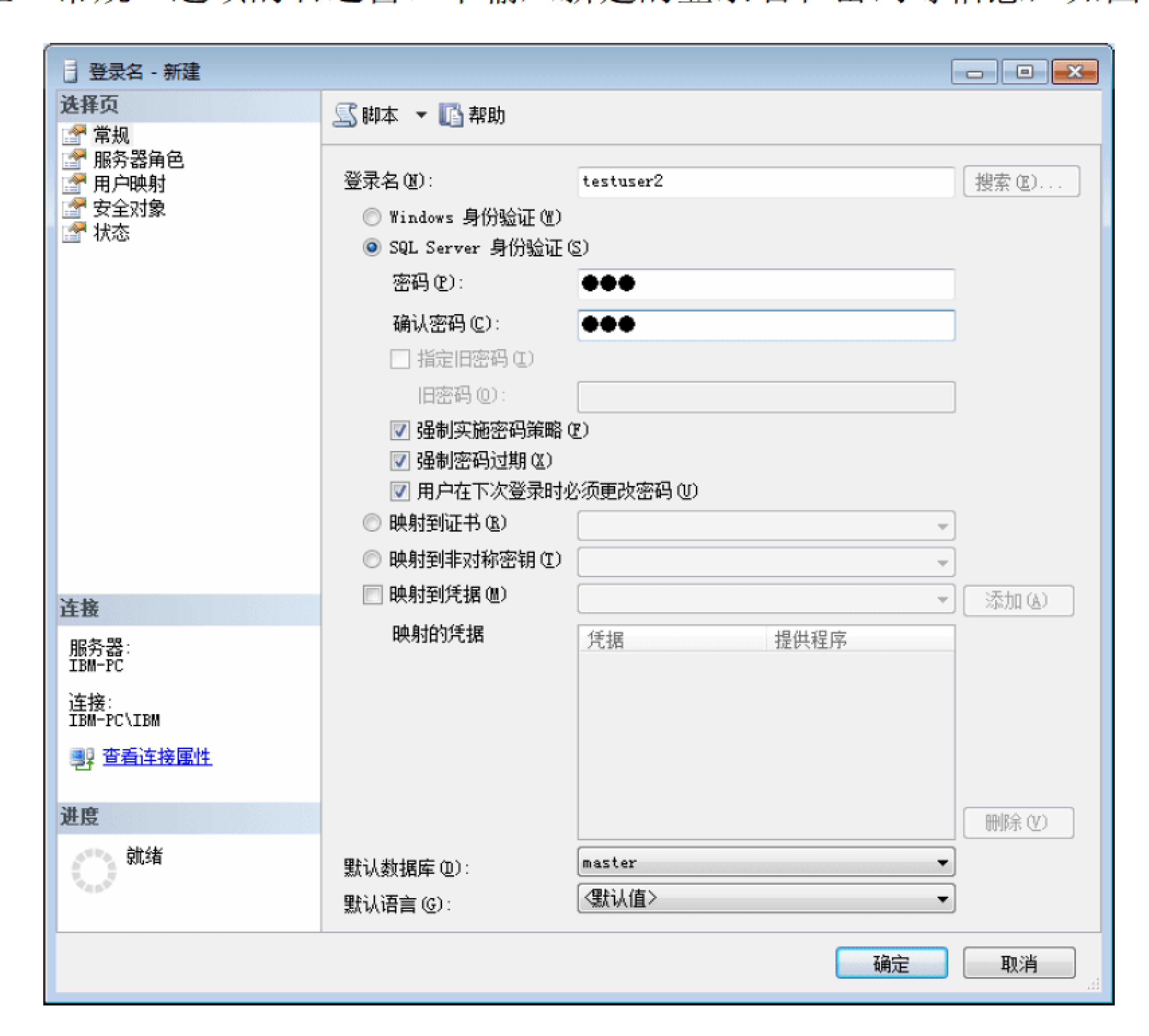


图 5.18　新建登录名

（4）在“用户映射”选项中选中 AdventureWorks2012 数据库左边的映射复选框，然后将“用户”列的值 testuser2 修改为 user2，这就是要创建的对应的用户，默认架构为 dbo，如图 5.19 所示。

（5）单击“确定”按钮，系统将创建登录名 testuser2 和在 AdventureWorks 数据库中对应的用户 user2，在对象资源管理器中将看到创建的用户，如图 5.20 所示。

（6）在其中右击 user2，在弹出的快捷菜单中选择“属性”选项，系统将弹出“数据库用户-user2”对话框。

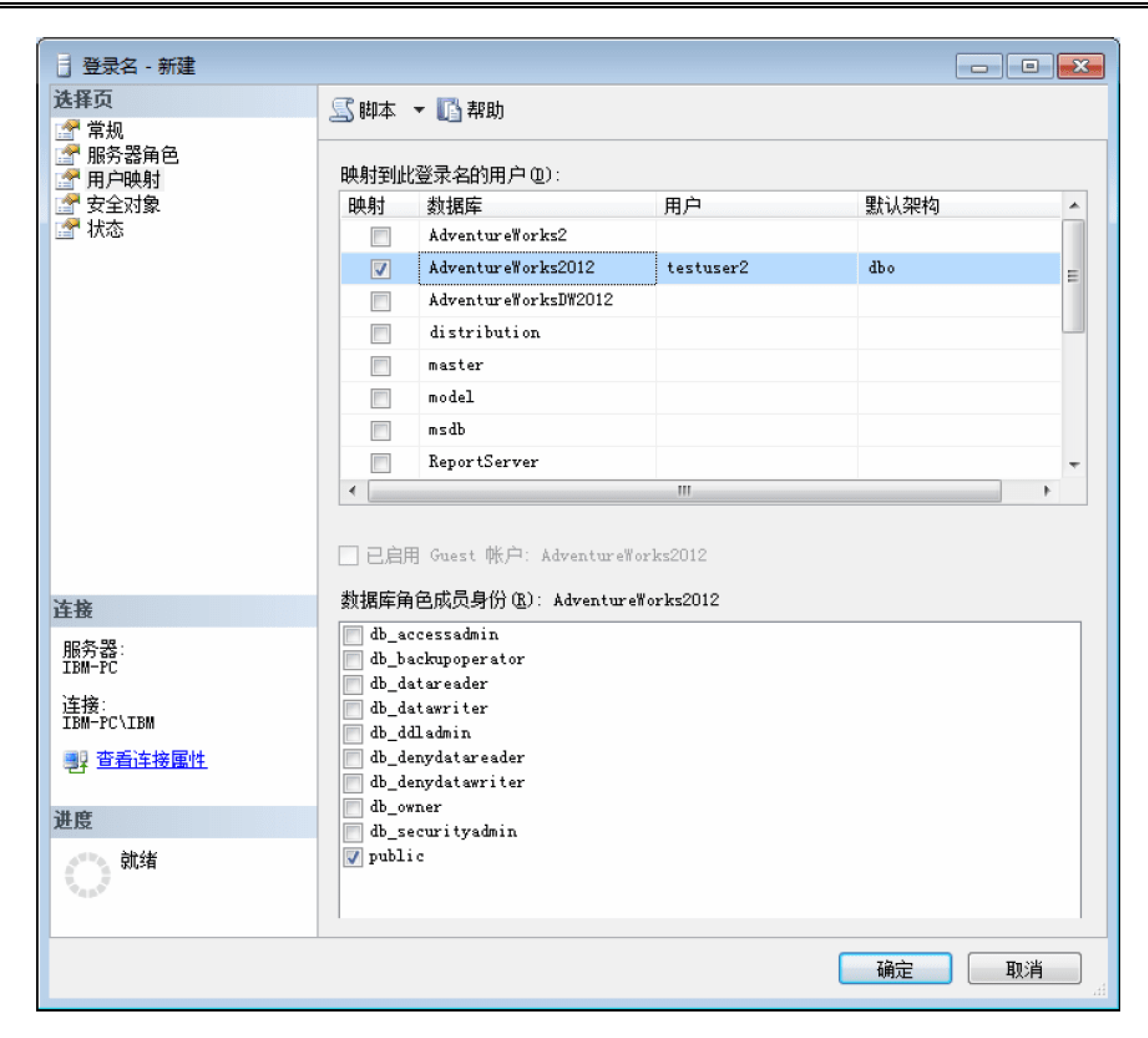


图 5.19　用户映射

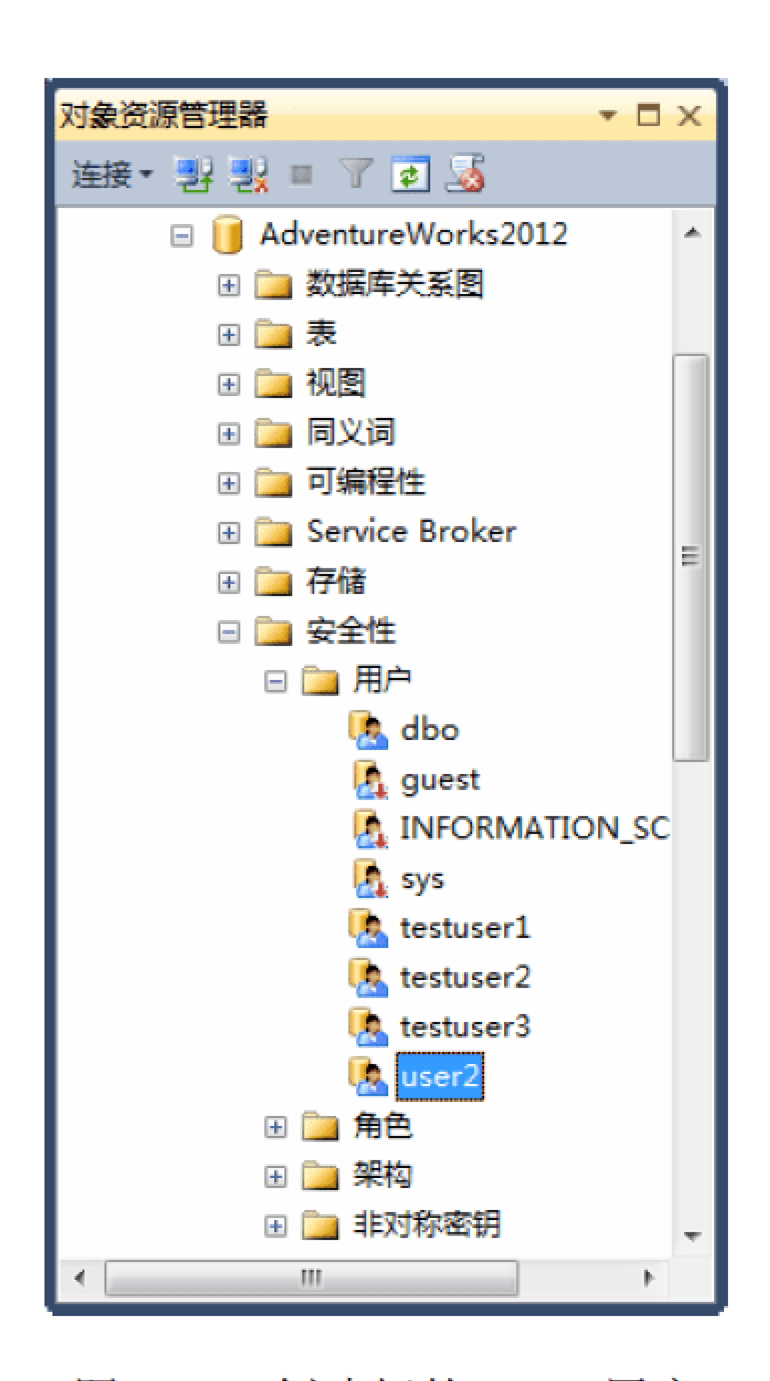


图 5.20　创建好的 user2 用户

（7）选择“安全对象”选项，切换到用户权限配置对话框，该对话框中显示了当前用户拥有的权限，如图 5.21 所示。其中并没有显示任何内容，是因为刚创建的用户并没有授予任何权限。

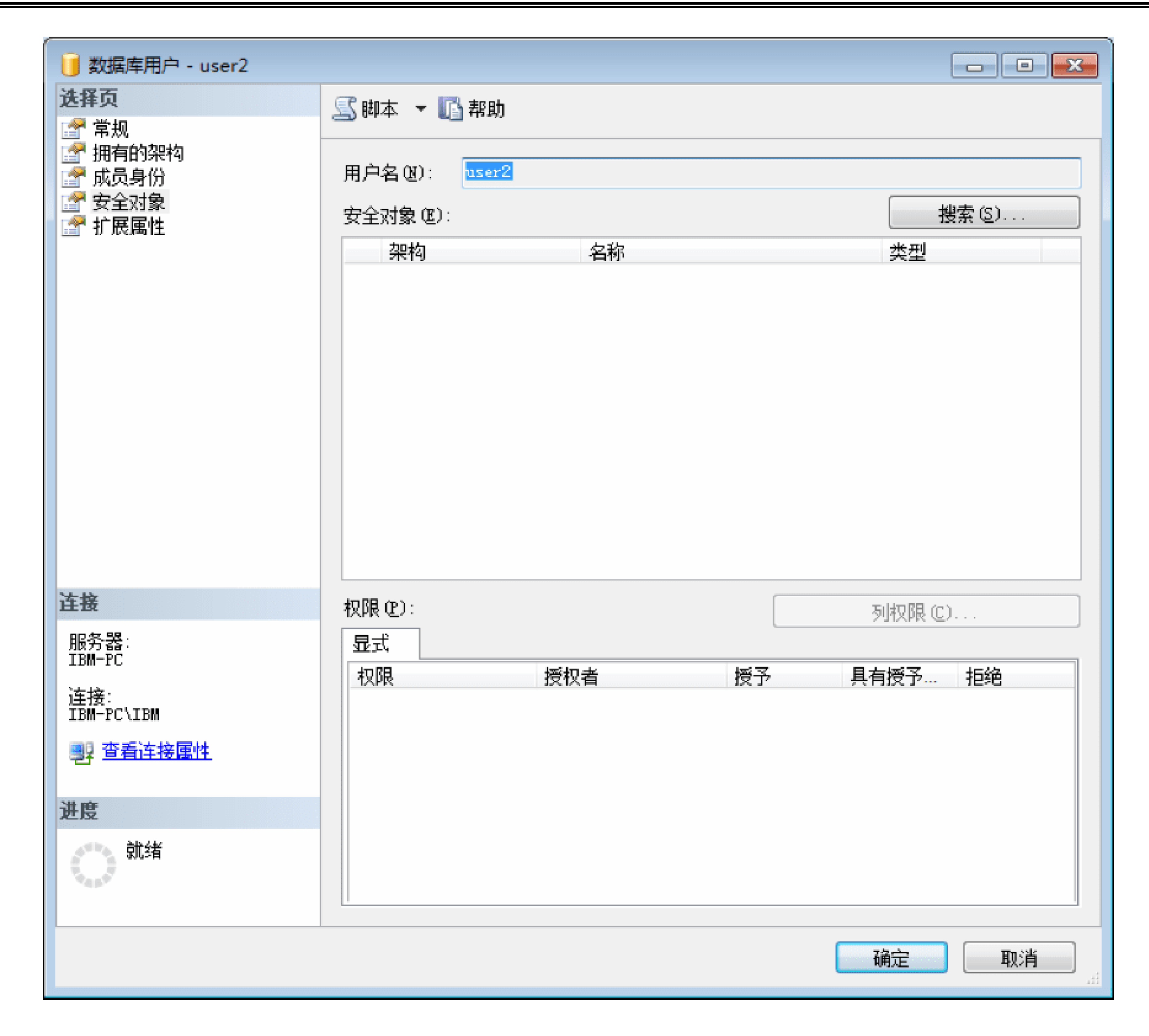


图 5.21　用户权限配置

（8）单击“搜索”按钮，系统将弹出“添加对象”对话框，如图 5.22 所示。该对话框中的“特定对象”单选按钮主要用于查找选择一个架构、一个表和一个存储过程等；“特定类型的所有对象”单选按钮就是按照类型分类，将一种类型的所有对象列出，主要用于多个同类型对象的权限操作。而“属于该架构的所有对象”单选按钮用于更快速地选出架构对象，这里选中“特定对象”单选按钮。



图 5.22　“添加对象”对话框

（9）单击“确定”按钮，系统弹出“选择对象”对话框，如图 5.23 所示。

（10）单击“对象类型”按钮，系统弹出“选择对象类型”对话框，该窗口列出了所有的对象类型，如图 5.24 所示。

（11）选中“架构”复选框，单击“确定”按钮，系统回到图 5.23 所示的对话框。在

该对话框中单击“浏览”按钮，系统弹出“查找对象”对话框，该对话框列出了当前数据库中的所有架构，如图 5.25 所示。



图 5.23　“选择对象”对话框

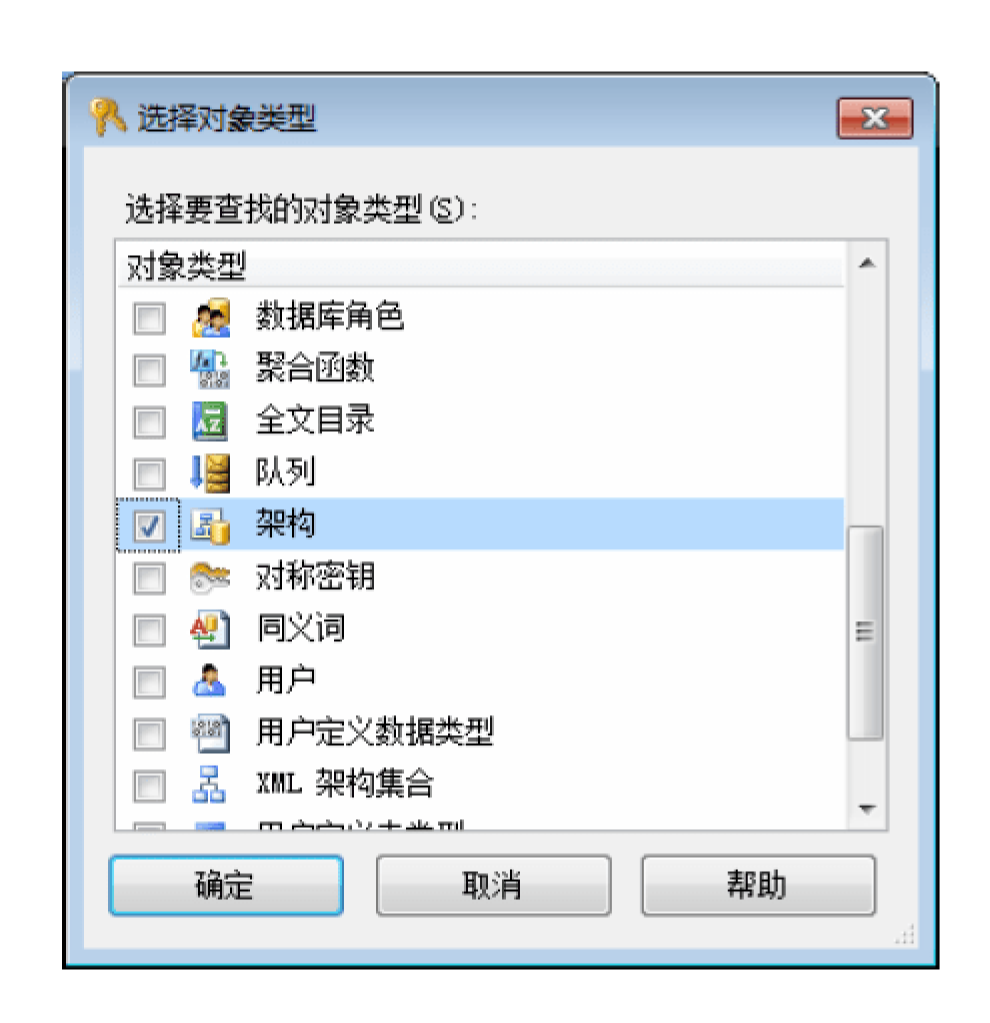


图 5.24　“选择对象类型”对话框

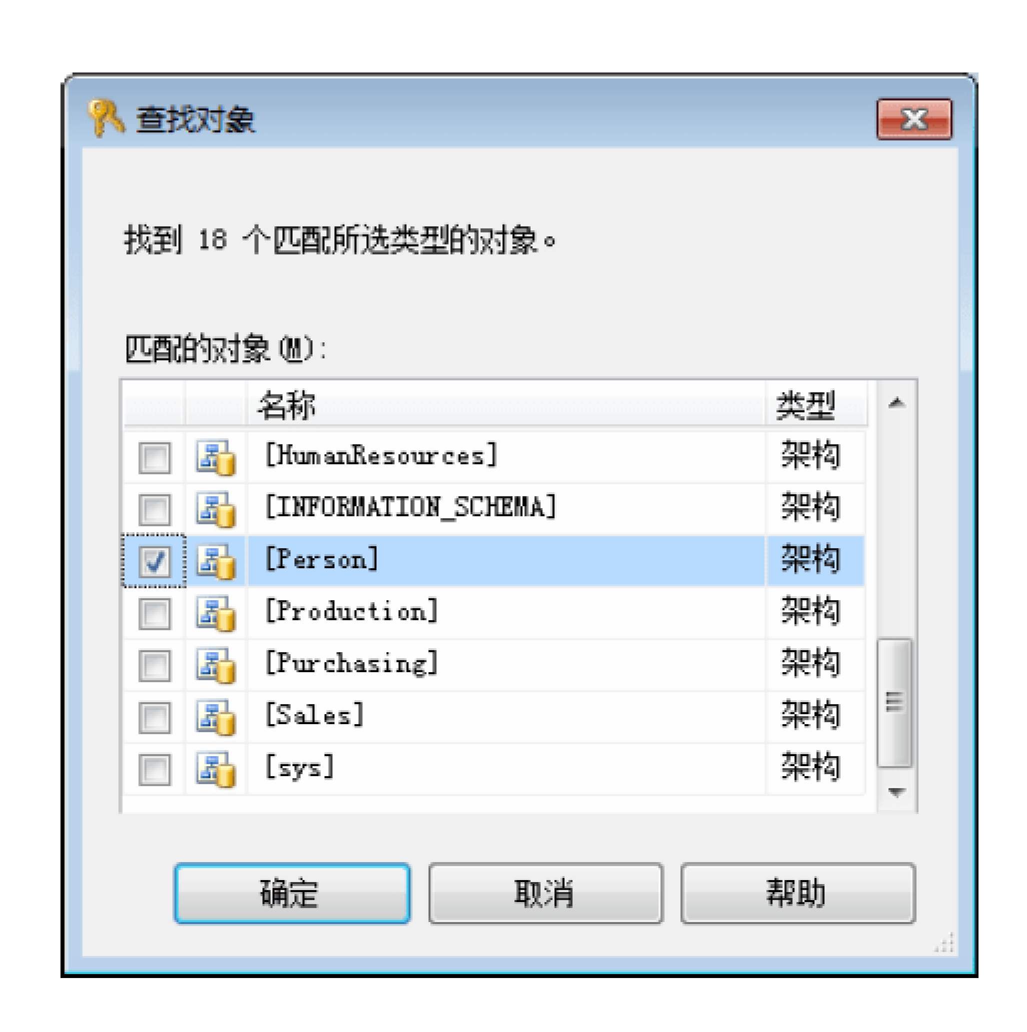


图 5.25　“查找对象”对话框

（12）选中[Person]复选框，然后单击“确定”按钮，系统回到图 5.23 所示的对话框。在该对话框单击“确定”按钮，系统回到数据库用户权限设置的对话框，如图 5.26 所示。

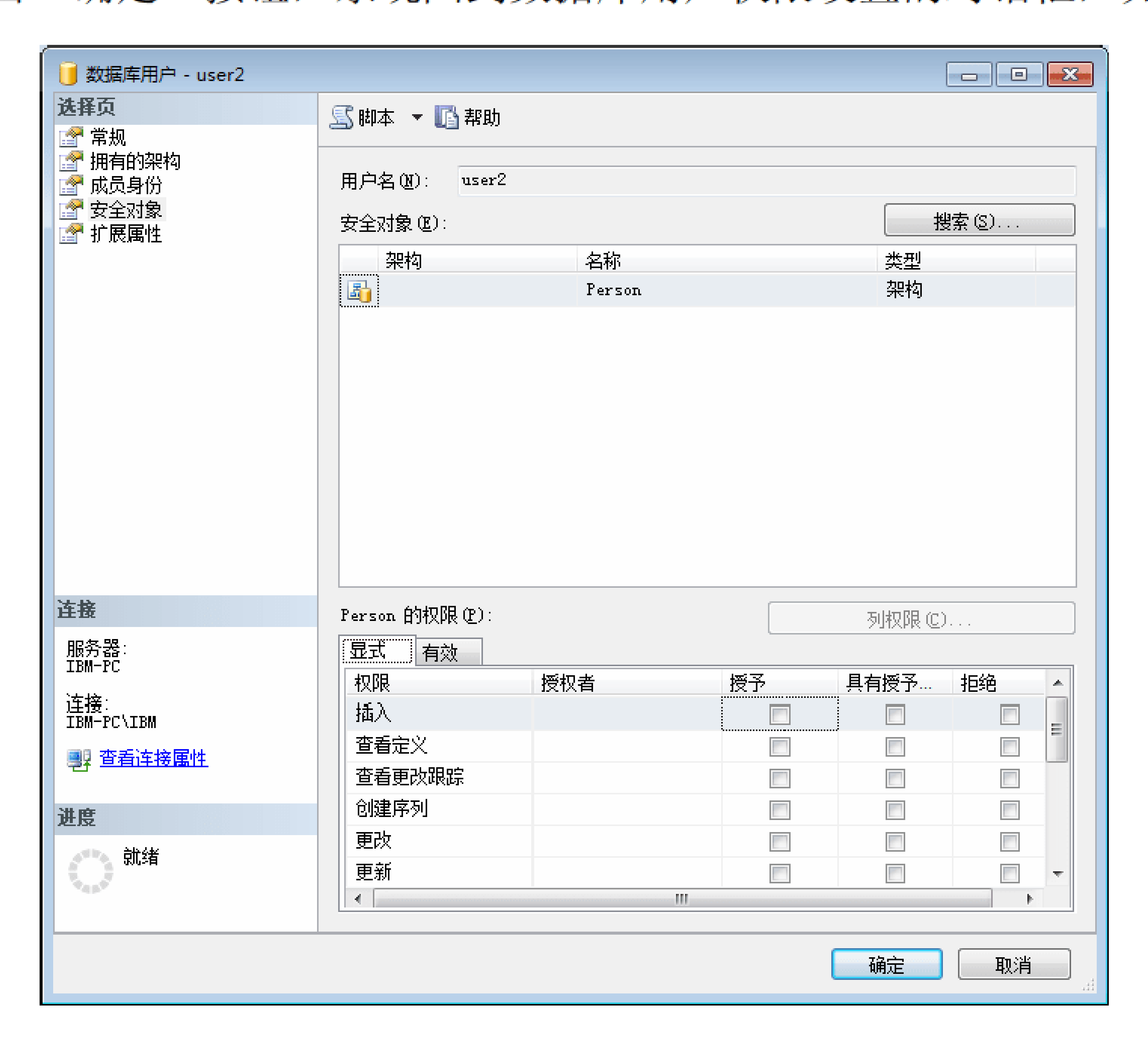


图 5.26　用户权限设置

（13）这里需要为 Person 架构设置 SELECT 权限，所以在“显式”选项卡的权限列表中将 Select 权限的授予复选框选中。

（14）再次单击“搜索”按钮，用同样的方法找到 Person.AddressType 表，然后将该表的 Update 权限的授予复选框选中，如图 5.27 所示。

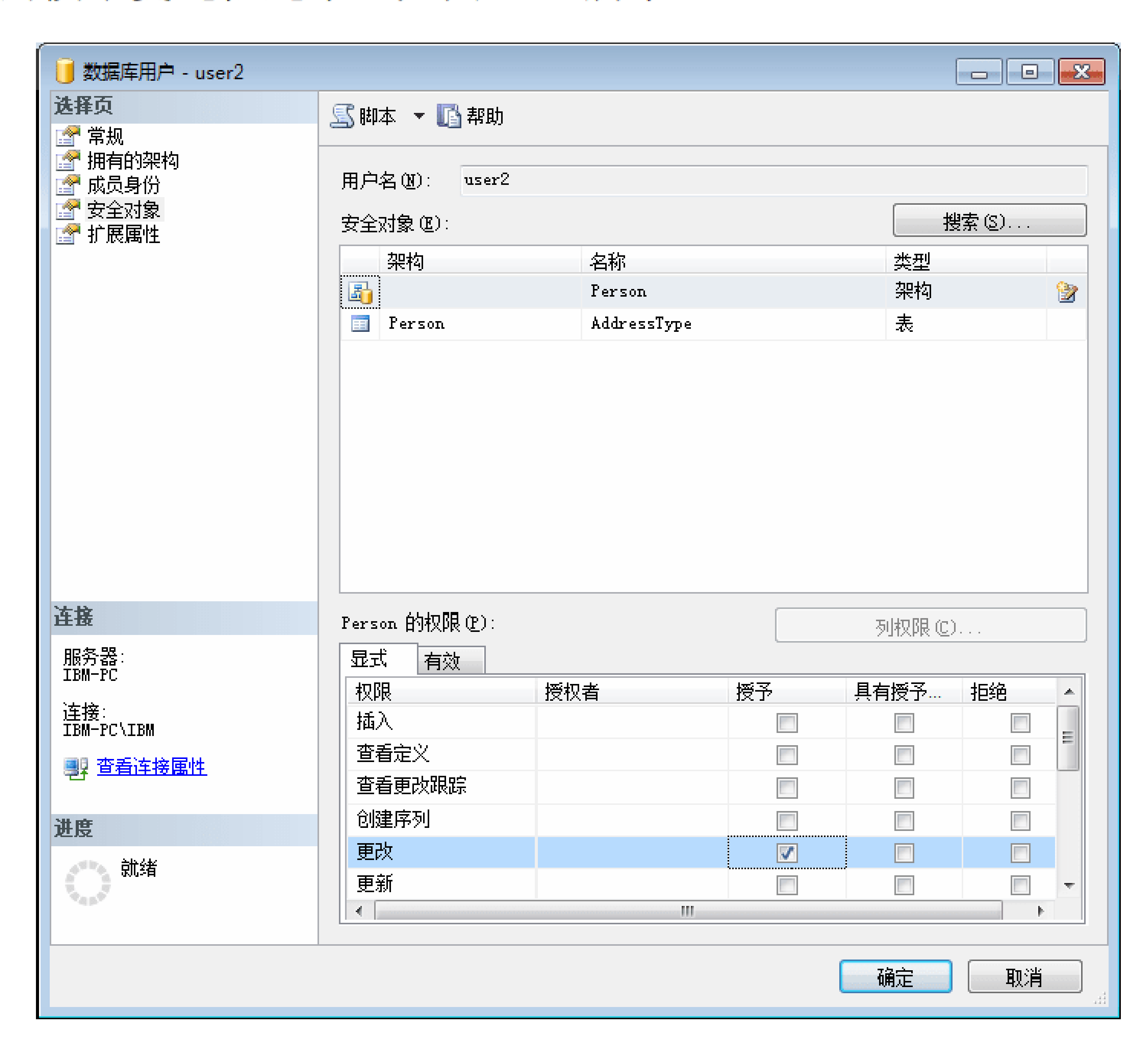


图 5.27　授予用户 Update 权限

（15）单击“确定”按钮，用户 user2 将具有授予的权限。

通过以上操作，登录名 testuser2 对应的 user2 将对 Person 架构的表具有 SELECT 权限，对 Person.AddressType 具有 UPDATE 权限。若现在需要对权限进行修改，使 user2 用户不再对 Person.AddressType 具有 UPDATE 权限，只需要重新打开 user2 的权限设置窗口，不选中 Person.AddressType 的 Update 授予选项即可，这相当于执行 REVOKE 命令。要显式拒绝对该表的 Update 权限，则选中 Update 拒绝选项即可，这相当于执行 DENY 命令。

另外，语句执行权限并不在用户的属性中进行设置，而是在登录名的属性中进行设置。例如要将创建数据库的权限授予 testuser2 登录名则对应的操作为：

（1）在对象资源管理器中展开“安全性”节点下的“登录名”节点。

（2）右击 testuser2 节点，在弹出的快捷菜单中选择“属性”选项，系统将弹出该登录名的属性对话框，如图 5.28 所示。

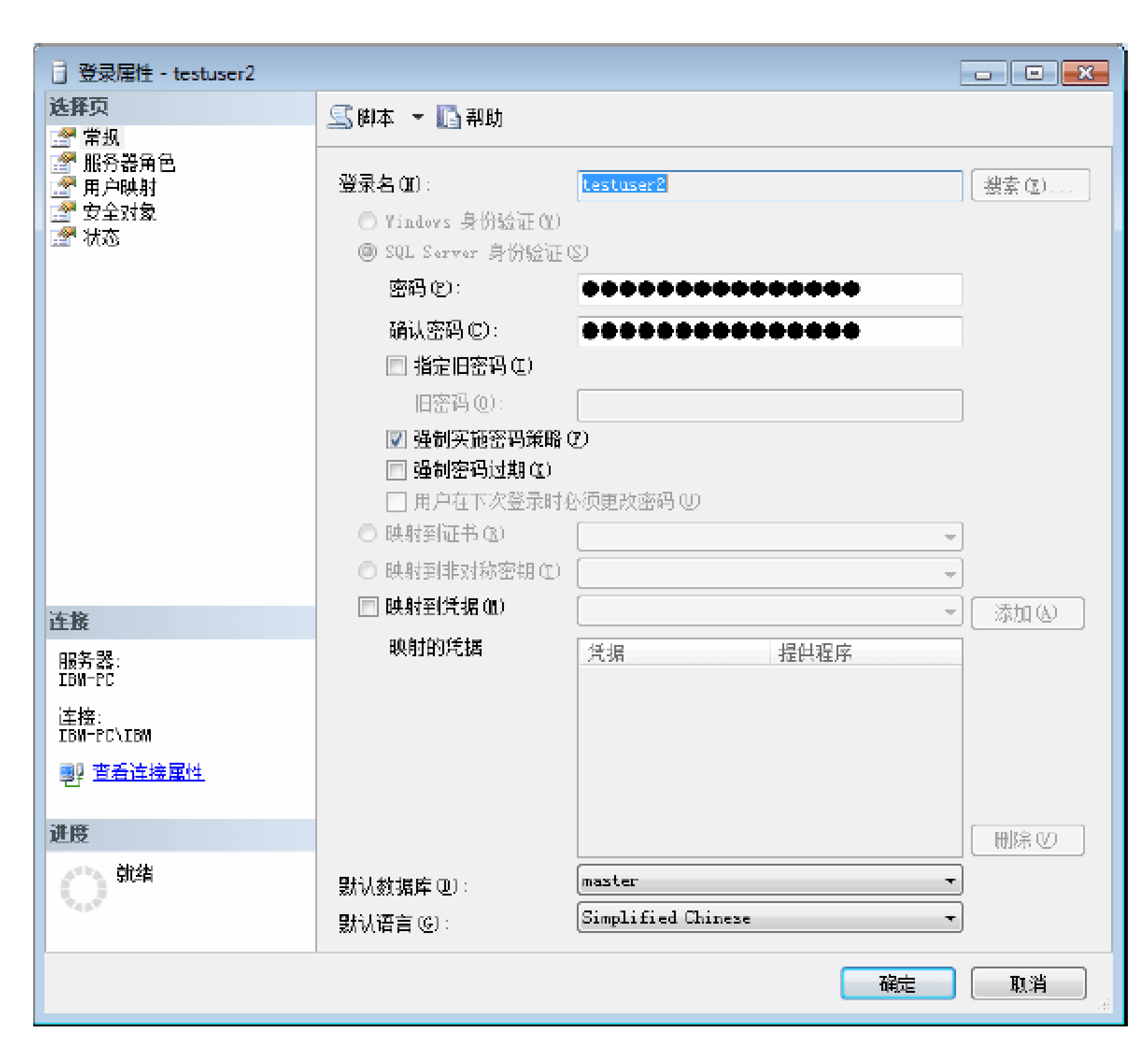


图 5.28　登录名的属性对话框

（3）选择“安全对象”选项，系统切换到对登录名的权限配置窗口，如图 5.29 所示。

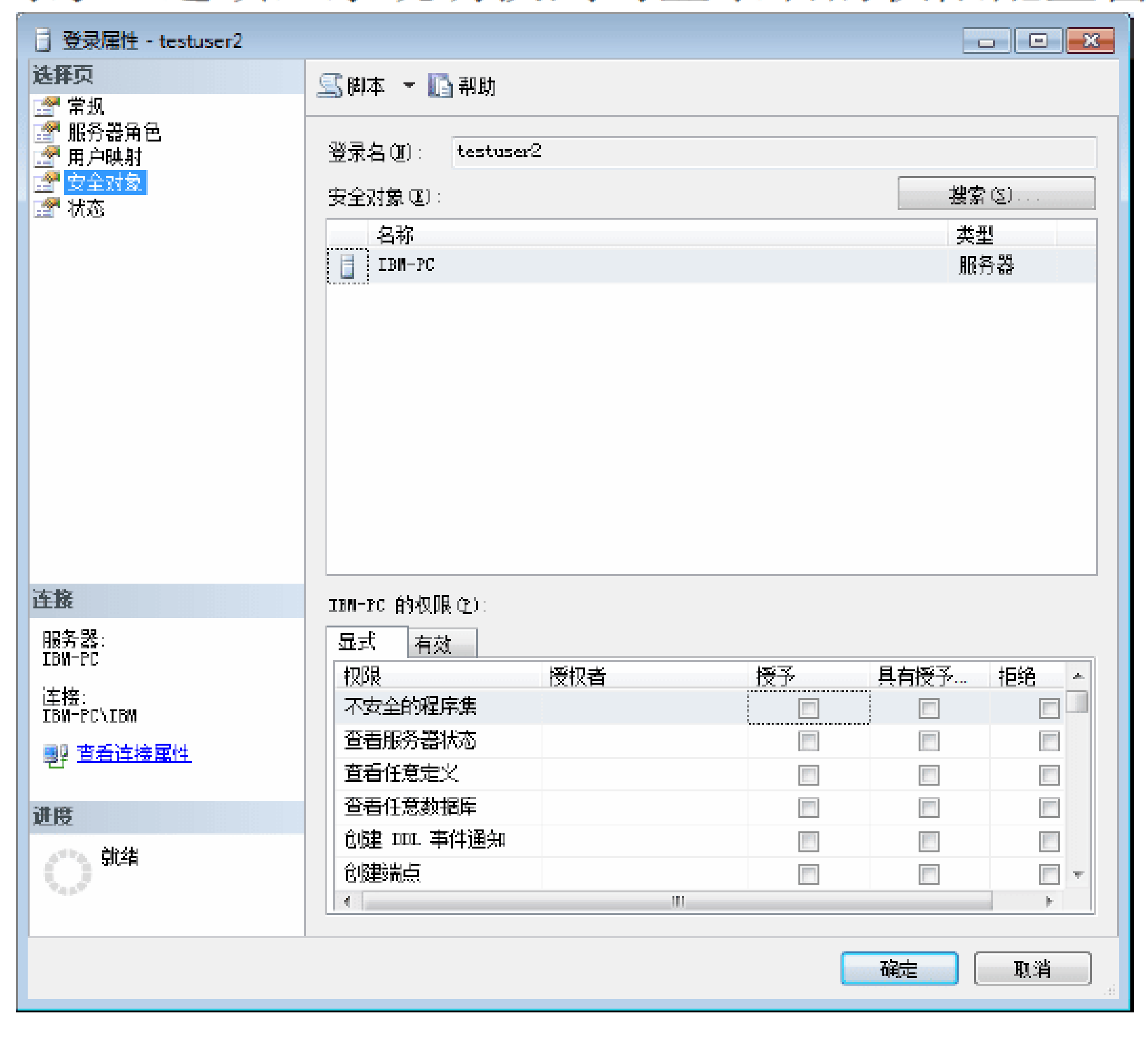


图 5.29　登录名安全对象窗口

（4）用类似与为用户授予权限的方式，找到服务器（这里服务器名为 IBM-PC），然后将“创建任意数据库”权限的授予复选框选中，如图 5.30 所示。

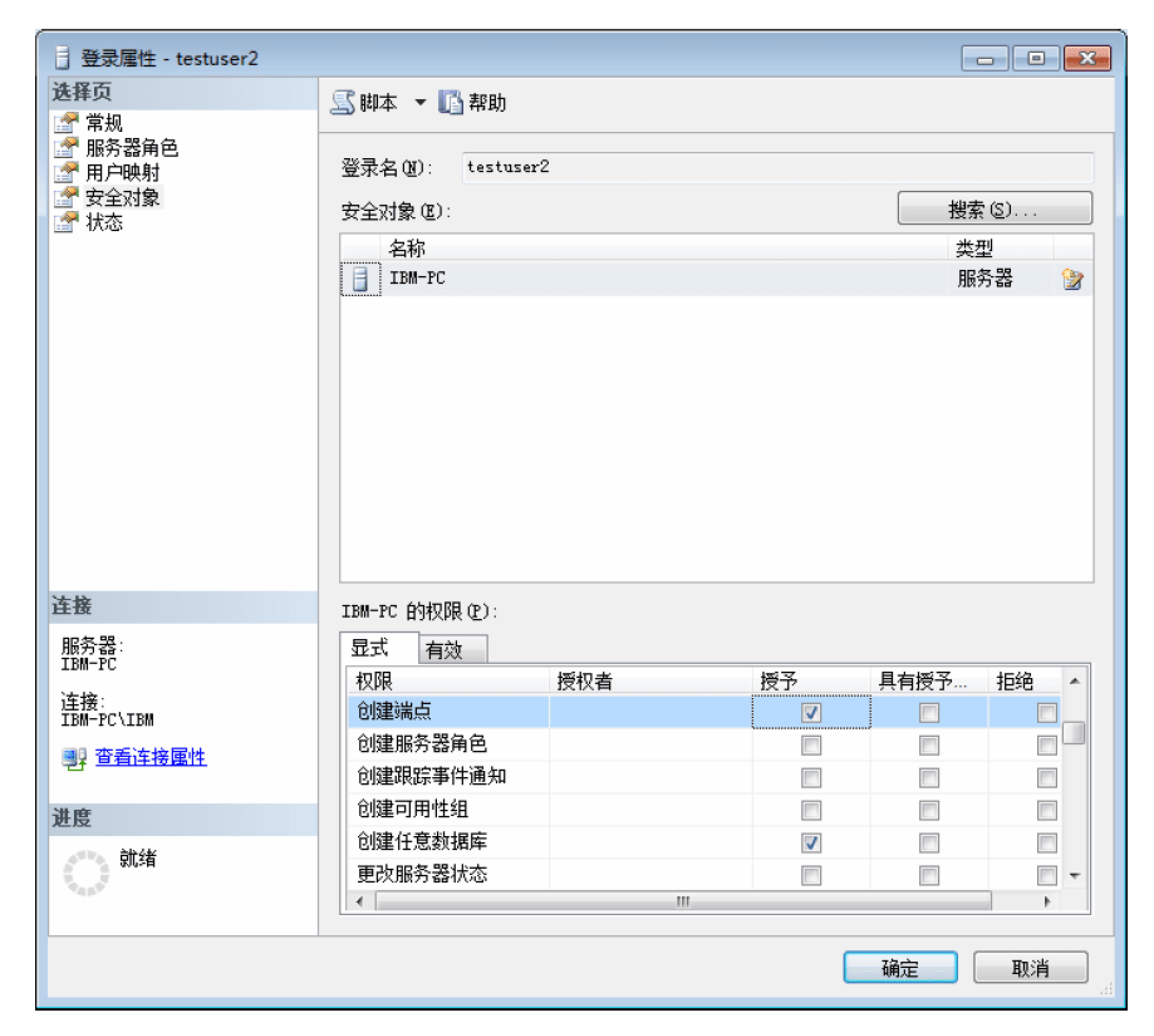


图 5.30　为登录名授予创建数据库权限

（5）单击“确定”按钮，系统将授予 testuser2 创建数据库的权限。

5.7　角 色 管 理

无论是在操作系统、一般业务软件系统还是在数据库管理中，角色都是一个很重要的概念，角色的出现极大地简化了权限管理。本节将主要讲解数据库中角色的使用。

5.7.1　角色简介

角色是一个访问权限的集合，只要给用户分配一个角色，就可以给这个用户全部分配这个权限集合。角色类似于 Windows 操作系统中的工作组的概念。

一个用户可以同时拥有多个角色。因为可以把用户访问权限分成更小的和更合逻辑的组并混合组成更适合用户的规则，所以角色极大地简化了权限的分配管理操作。角色分为两类：服务器角色和数据库角色。

除了这两种角色类型外，SQL Server 2012 中还有一种角色被称为应用程序角色。应用程序角色是一个数据库主体，它使应用程序能够用其自身的、类似用户的权限来运行。

这其中服务器级角色也称为“固定服务器角色”，因为用户不能创建新的服务器级角色。服务器级角色的权限作用域为服务器范围。SQL Server 2012 中有两种类型的数据库级角色：数据库中预定义的“固定数据库角色”和可以创建的“用户定义数据库角色”。固定数据库角色是在数据库级别定义的，并且存在于每个数据库中。下面分别介绍这几种角色。

5.7.2　服务器角色

笔者在前面已经提到，服务器角色是固定的不可被用户创建的，用户在安装完成 SQL Server 2008 时所有的服务器角色就已经存在。用户可以向服务器级角色中添加 SQL Server 登录名、Windows 账户和 Windows 组。固定服务器角色的每个成员都可以向其所属角色添加其他登录名。SQL Server 2012 中常用的服务器角色和说明如表 5.2 所示。

表 5.2　服务器角色

固定服务器角色	服务器级权限	说　　明
bulkadmin	已授予：ADMINISTER BULK OPERATIONS	可以运行BULK INSERT批量插入语句
dbcreator	已授予：CREATE DATABASE	可以创建、更改、删除和还原任何数据库
Diskadmin	已授予：ALTER RESOURCES	用于管理服务器的磁盘文件
Processadmin	已授予：ALTER ANY CONNECTION、ALTER SERVER STATE	可以终止在SQL Server 实例中运行的进程
securityadmin	已授予：ALTER ANY LOGIN	可以管理实例中的登录名及其属性
serveradmin	已授予：ALTER ANY ENDPOINT、ALTER RESOURCES、ALTER SERVER STATE、ALTER SETTINGS、SHUTDOWN、VIEW SERVER STATE	可以更改服务器范围的配置选项和关闭服务器
setupadmin	已授予：ALTER ANY LINKED SERVER	可以在实例中添加和删除链接服务器
sysadmin	已使用GRANT选项授予：CONTROL SERVER	超级权限，可以在服务器上执行任何活动

如果将服务器角色赋予登录名，则需要使用系统存储过程 sp_addsrvrolemember。该存储过程的语法为：

```
sp_addsrvrolemember [ @loginame= ] 'login' , [ @rolename = ] 'role'
```

其中[@loginame=]'login'是添加到固定服务器角色中的登录名。login 的数据类型为 sysname，无默认值。login 可以是 SQL Server 登录或 Windows 登录。如果未向 Windows 登录授予对 SQL Server 的访问权限，则将自动授予该访问权限。

[@rolename=]'role'是要添加登录的固定服务器角色的名称。role 的数据类型为 sysname，默认值为 NULL，且必须为固定服务器角色中的一个。例如现在有登录名 testuser1，要赋予该登录名 dbcreator 的服务器角色，那么对应的 SQL 脚本为：

```
EXEC sp_addsrvrolemember 'testuser1','dbcreator'
```

对应于赋予用户角色的 sp_addsrvrolemember 命令，SQL Server 2012 同样提供了 sp_dropsrvrolemember 命令，用于从服务器角色中删除 SQL 登录名或 Windows 用户或组。

sp_dropsrvrolemember 的语法为：

```
sp_dropsrvrolemember [ @loginame = ] 'login' , [ @rolename = ] 'role'
```

各参数的含义与 sp_addsrvrolemember 的参数含义相同，这里就不再重复介绍。例如，要将刚为 testuser1 添加的 dbcreator 角色去掉，那么对应的 SQL 脚本为：

```
EXEC sp_dropsrvrolemember 'testuser1','dbcreator'
```

在 SSMS 中，对登录名或 Windows 用户或组的服务器角色操作也十分方便，主要操作步骤如下所述。

（1）在 SSMS 的对象资源管理器中展开“安全性”节点下的“登录名”节点。

（2）双击需要配置服务器角色的登录名或者右击该登录名，在弹出的快捷菜单中选择“属性”选项，系统将弹出“登录属性”对话框。

（3）单击“服务器角色”选项，系统将切换到服务器角色配置的窗口，如图 5.31 所示。

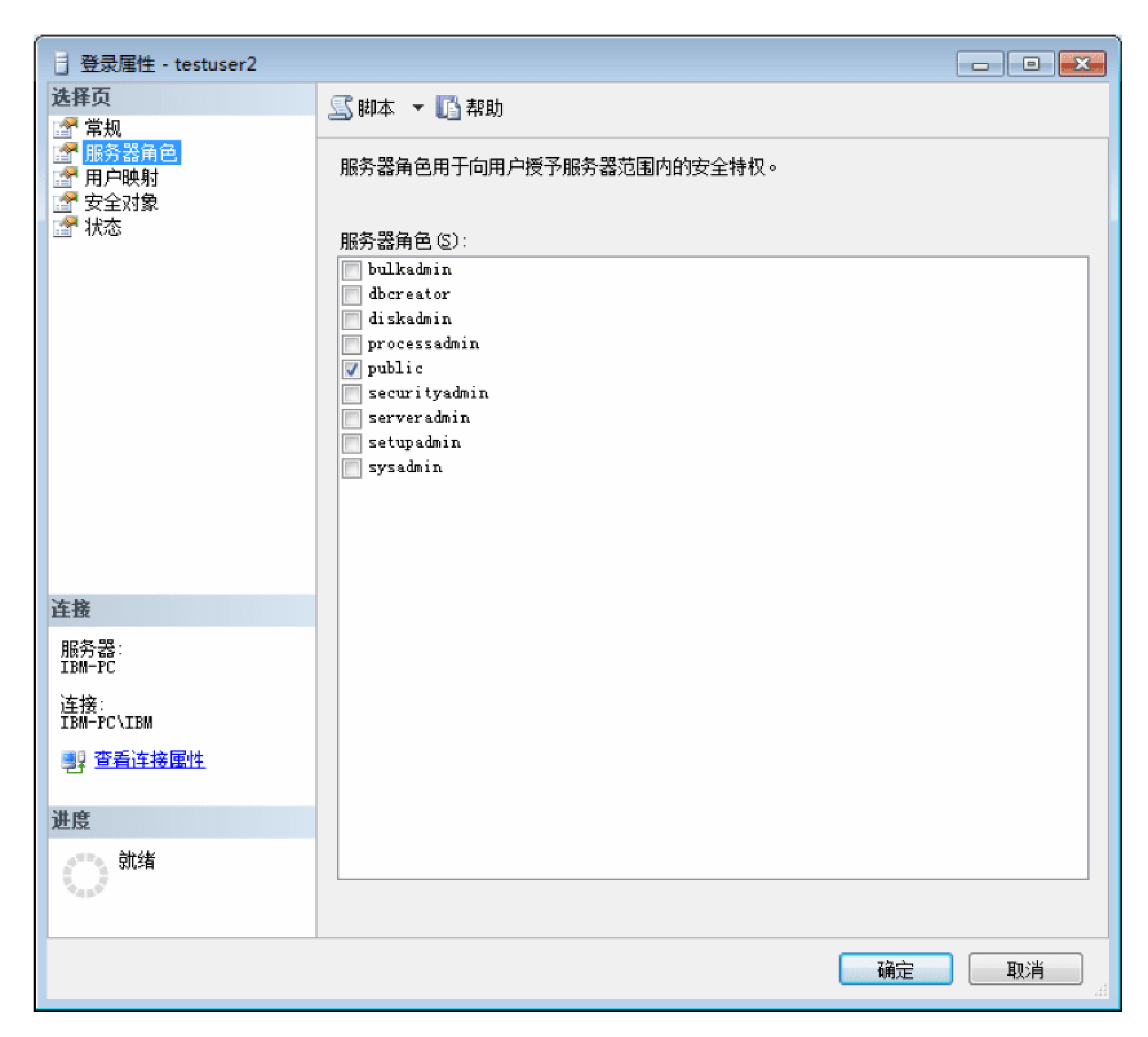


图 5.31　服务器角色配置

（4）选择需要赋予的角色，然后单击“确定”按钮即可完成服务器角色的配置。

注意：在 SSMS 中看到服务器角色中有 public 这样一个角色，但是该角色其实并不是服务器角色，而是公共角色，不能通过 sp_dropsrvrolemember 命令取消登录名的 public 角色。public 角色拥有 VIEW ANY DATABASE 权限。

5.7.3　固定数据库角色

与固定服务器角色类似，SQL Server 2012 中也提供了固定的数据库角色。固定数据库角色主要是为了简化权限配置过程，所以大部分固定的数据库角色其实可以通过用户定义的数据库角色来实现，但是仍有部分固定数据库角色是不可替代的。如表 5.3 列出了固定数据库角色对应的权限。

表 5.3　固定数据库角色的权限

固定数据库角色	数据库级权限	服务器级权限
db_accessadmin	已授予：ALTER ANY USER、CREATE SCHEMA	已授予：VIEW ANY DATABASE
db_accessadmin	已使用GRANT选项授予：CONNECT	无
db_backupoperator	已授予：BACKUP DATABASE、BACKUP LOG、CHECKPOINT	已授予：VIEW ANY DATABASE
db_datareader	已授予：SELECT	已授予：VIEW ANY DATABASE
db_datawriter	已授予：DELETE、INSERT、UPDATE	已授予：VIEW ANY DATABASE
db_ddladmin	已授予：ALTER ANY ASSEMBLY、ALTER ANY ASYMMETRIC KEY、ALTER ANY CERTIFICATE、ALTER ANY CONTRACT、ALTER ANY DATABASE DDL TRIGGER、ALTER ANY DATABASE EVENT、NOTIFICATION、ALTER ANY DATASPACE、ALTER ANY FULLTEXT CATALOG、ALTER ANY MESSAGE TYPE、ALTER ANY REMOTE SERVICE BINDING、ALTER ANY ROUTE、ALTER ANY SCHEMA、ALTER ANY SERVICE、ALTER ANY SYMMETRIC KEY、CHECKPOINT、CREATE AGGREGATE、CREATE DEFAULT、CREATE FUNCTION、CREATE PROCEDURE、CREATE QUEUE、CREATE RULE、CREATE SYNONYM、CREATE TABLE、CREATE TYPE、CREATE VIEW、CREATE XML SCHEMA COLLECTION、REFERENCES	已授予：VIEW ANY DATABASE
db_denydatareader	已拒绝：SELECT	已授予：VIEW ANY DATABASE
db_denydatawriter	已拒绝：DELETE、INSERT、UPDATE	无
db_owner	已使用GRANT选项授予：CONTROL	已授予：VIEW ANY DATABASE
db_securityadmin	已授予：ALTER ANY APPLICATION ROLE、ALTER ANY ROLE、CREATE SCHEMA、VIEW DEFINITION	已授予：VIEW ANY DATABASE

如表 5.4 给出了每个数据库角色的说明。

表 5.4　固定数据库角色说明

服务器级角色名称	说　　明
db_owner	数据库所有者，可以执行数据库的所有配置和维护活动，还可以删除数据库
db_securityadmin	安全相关管理，可以修改角色成员身份和管理权限。向此角色中添加主体可能会导致意外的权限升级
db_accessadmin	访问管理，可以为Windows登录名、Windows组和SQL Server登录名添加或删除数据库访问权限
db_backupoperator	备份管理角色，可以备份数据库
db_ddladmin	数据定义管理，可以在数据库中运行任何数据定义语言DDL命令
db_datawriter	可修改数据，可以在所有用户表中添加、删除或更改数据
db_datareader	只读角色，可以从所有用户表中读取所有数据
db_denydatawriter	不能添加、修改或删除数据库内用户表中的任何数据
db_denydatareader	不能读取数据库内用户表中的任何数据

要将数据库角色赋予数据库用户或者 Windows 用户或组，SQL Server 2012 提供了系统存储过程 sp_addrolemember。该存储过程的语法为：

```
sp_addrolemember [ @rolename = ] 'role',[ @membername = ] 'security_account'
```

其中，[@rolename=]'role'为当前数据库中的数据库角色名称。role 数据类型为 sysname，无默认值。[@membername=]'security_account'是添加到该角色的安全账户。security_account 数据类型为 sysname，无默认值。security_account 可以是数据库用户、数据库角色、Windows 登录或 Windows 组。

例如在 AdventureWorks2012 数据库中有数据库用户 test1，现在希望该用户能够以只读的方式访问该数据库，那么可以为该用户赋予 db_datareader 角色。具体 SQL 脚本如代码 5.29 所示。

代码 5.29　赋予 test1 用户 db_datareader 角色

```
USE [AdventureWorks2012]
GO
EXEC sp_addrolemember 'db_datareader', 'test1' --为用户添加角色
GO
```

除了为用户添加角色外，SQL Server 2012 也提供了系统存储过程 sp_droprolemember，用于将用户从角色中删除。该存储过程的语法为：

```
sp droprolemember [ @rolename = ] 'role' , [ @membername = ]
'security_account'
```

各参数含义与 sp_addrolemember 相同，这里就不再重述。例如需要将 AdventureWorks 数据库中的数据库用户 test1 从 db_datareader 角色中删除，那么对应的 SQL 脚本如代码 5.30 所示。

代码 5.30　删除 test1 用户的 db_datareader 角色

```
USE [AdventureWorks2012]
GO
EXEC sp_droprolemember 'db_datareader', 'test1' --删除 test1 用户的 db_
```

```
datareader 角色
GO
```

同样以 AdventureWorks2012 数据库中的 test1 为例，在 SSMS 中要向固定数据库角色添加或删除用户的主要操作如下所述。

（1）在 SSMS 的对象资源管理器中依次展开数据库、AdventureWorks2012、安全、用户等节点。

（2）双击 test1 用户节点或右击该节点，在弹出的快捷菜单中选择“属性”选项，系统将弹出“数据库用户”对话框，如图 5.32 所示。

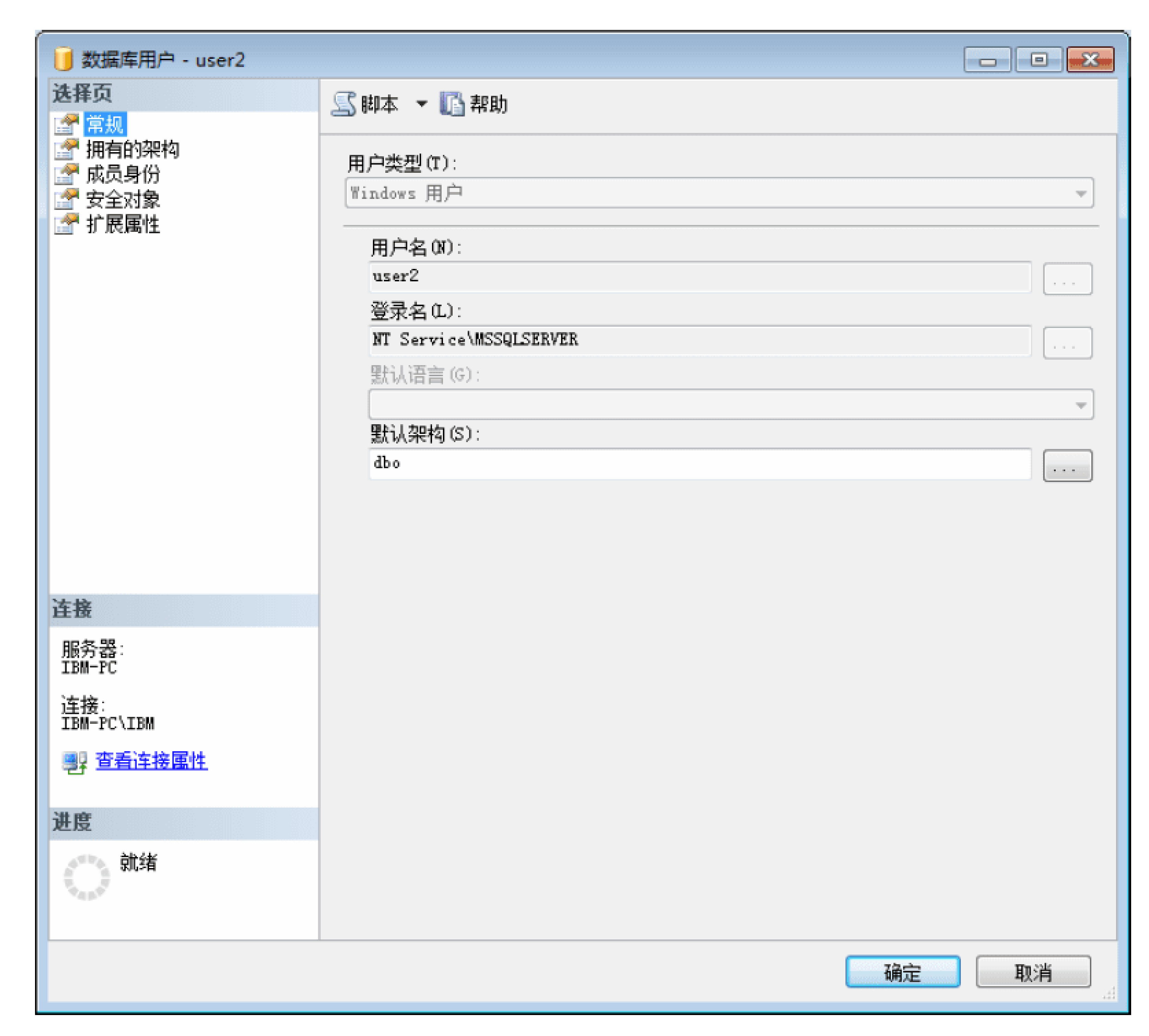


图 5.32　“数据库用户”对话框

（3）在该对话框单击“成员身份”选项，如图 5.33 所示。列出了当前用户所能拥有的数据库角色，选中要赋予的角色，取消选中不需要赋予的角色。

（4）单击“确定”按钮，就完成了对用户数据库角色的配置。

5.7.4　用户定义数据库角色

固定的数据库角色有助于帮助用户快速地配置权限，但是安全性中的核心任务是创建和分配用户定义的数据库角色，这些角色可以决定它们包含什么样的许可权。

对于用户定义的数据库角色，可以用处理数据库用户的方法一样授权、拒绝和回收权限。使用角色进行权限配置，可以通过修改角色这一个地方而把权限修改应用到每一个配置该角色的用户上。要创建用户第一的数据库角色，SQL Server 2012 提供了 CREATE ROLE 命令，该命令的语法为：

```
CREATE ROLE role_name [ AUTHORIZATION owner_name ]
```

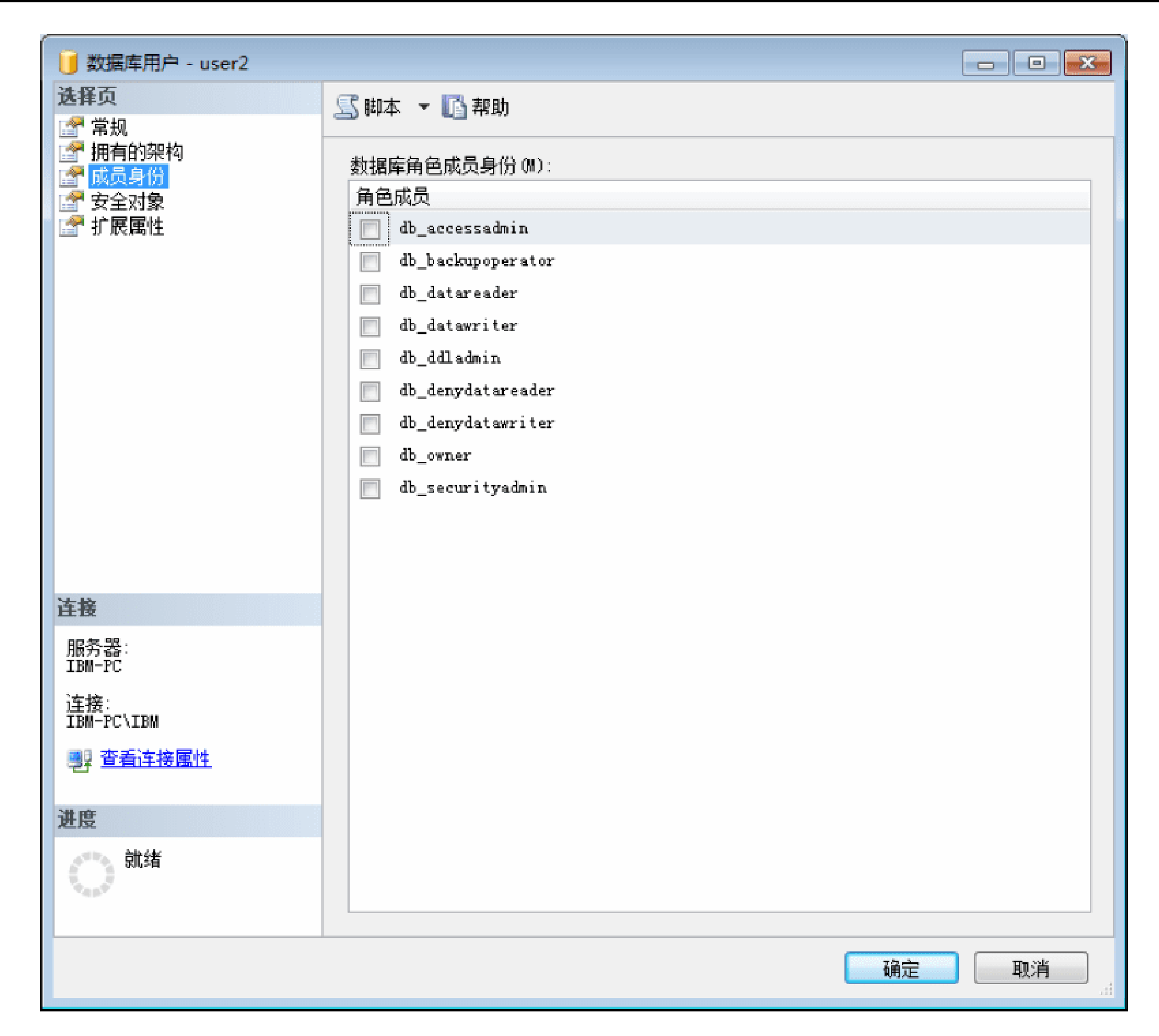


图 5.33 “成员身份”选项

其中 role_name 为待创建角色的名称。AUTHORIZATION owner_name 将拥有新角色的数据库用户或角色。如果未指定用户，则执行 CREATE ROLE 的用户将拥有该角色。例如要在 AdventureWorks2012 数据库中创建角色 HRReader，该角色拥有对 HumanResources 架构表的 SELECT 权限，那么对应的 SQL 语句如代码 5.31 所示。

代码 5.31 创建角色

```
USE AdventureWorks2012
GO
CREATE role HRReader --创建角色名
GO
GRANT SELECT ON SCHEMA::HumanResources  --角色权限
TO HRReader
```

角色创建后就需要将角色分配给具体的用户。给用户定义角色添加用户的操作与给固定数据库角色用户添加角色的方法是一样的，都是使用系统存储过程 sp_addrolemember。关于 sp_addrolemember 的语法和参数，在前面固定数据库角色章节已经做了详细介绍，这里就不重复介绍了。

要向刚建立的角色 HRReader 中添加用户 test1 的 SQL 脚本为：

```
EXEC sp_addrolemember 'HRReader','test1'
```

同样，若需要将角色中的用户删除时使用系统存储过程 sp_droprolemember。例如要将 test1 用户从 HRReader 角色中移除的 SQL 脚本为：

```
EXEC sp_droprolemember 'HRReader','test1'
```

使用 SSMS 也可以创建用户定义数据库角色。以在 AdventureWorks2012 数据库中创建

角色 SalesReader 为例，在 SSMS 中创建用户定义数据库角色的主要操作如下所述。

（1）在 SSMS 的对象资源管理器中依次展开数据库、AdventureWorks2012、安全性、角色、数据库角色节点。

（2）右击“数据库角色”节点，在弹出的快捷菜单中选择“新建数据库角色”选项，系统将弹出“数据库角色-新建”对话框，如图 5.34 所示。

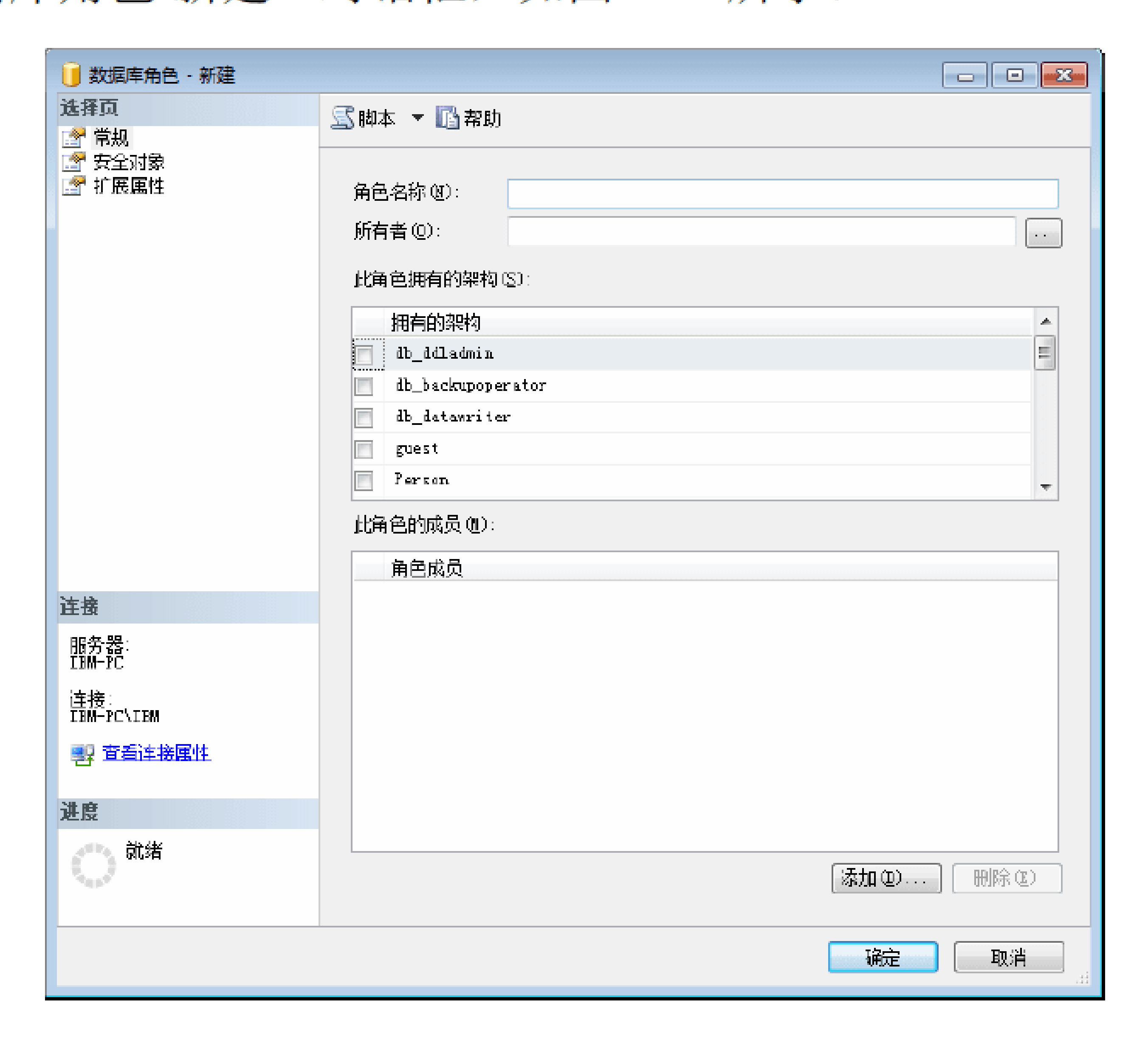


图 5.34　“数据库角色-新建”对话框

（3）在“角色名称”文本框中输入要新建的角色 SalesReader。

（4）单击“添加”按钮，选出要添加到该角色中的用户，例如 test1。

（5）选择“安全对象”选项，系统切换到角色的权限配置窗口，如图 5.35 所示。

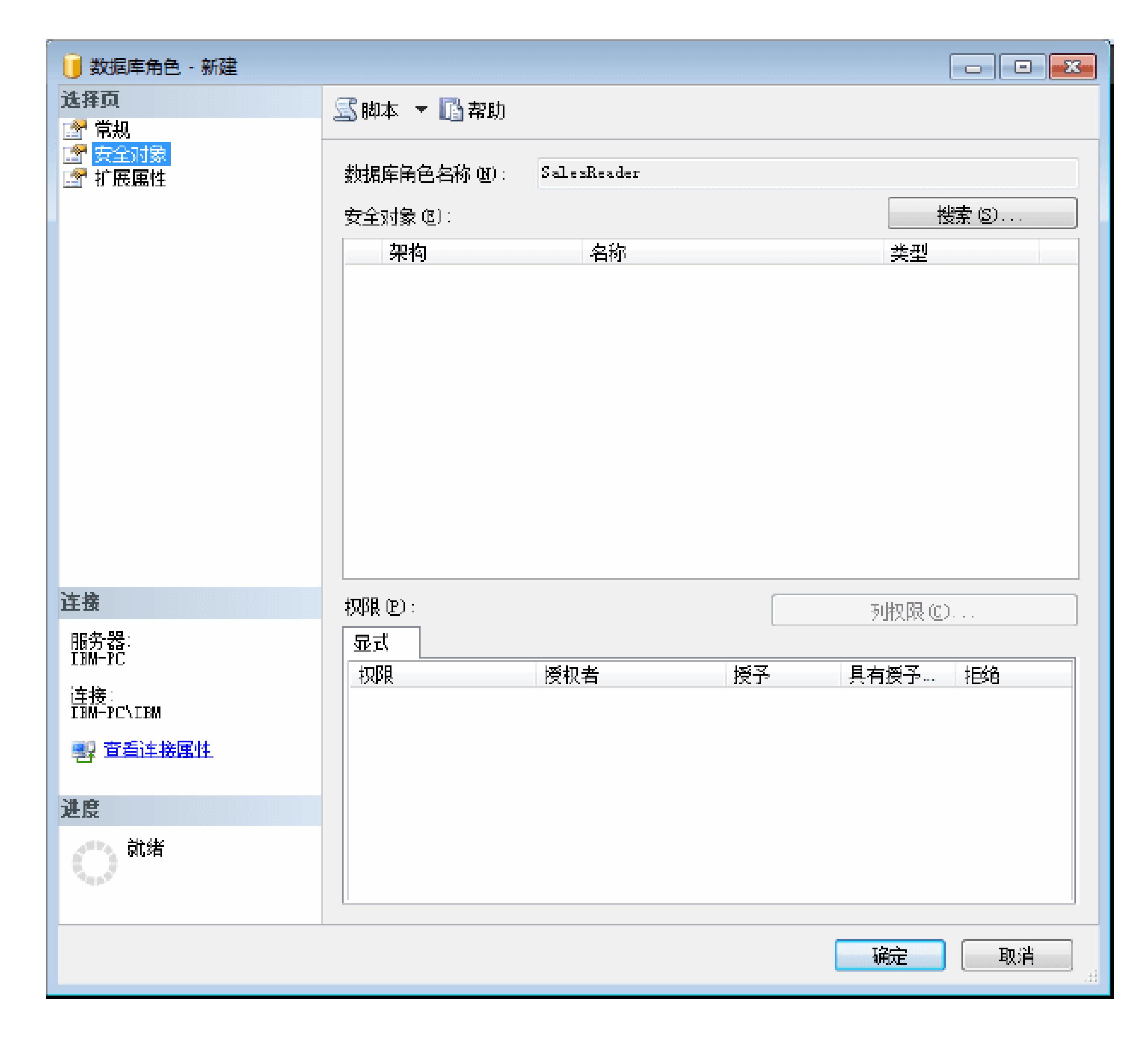


图 5.35　角色的权限配置窗口

（6）单击“搜索”按钮，接下来的操作与用户权限配置的操作相同，读者若不是很清

楚，可以查看 5.6.6 节中的内容。

（7）为该用户配置对 Sales 架构的选择权限，配置后如图 5.36 所示。

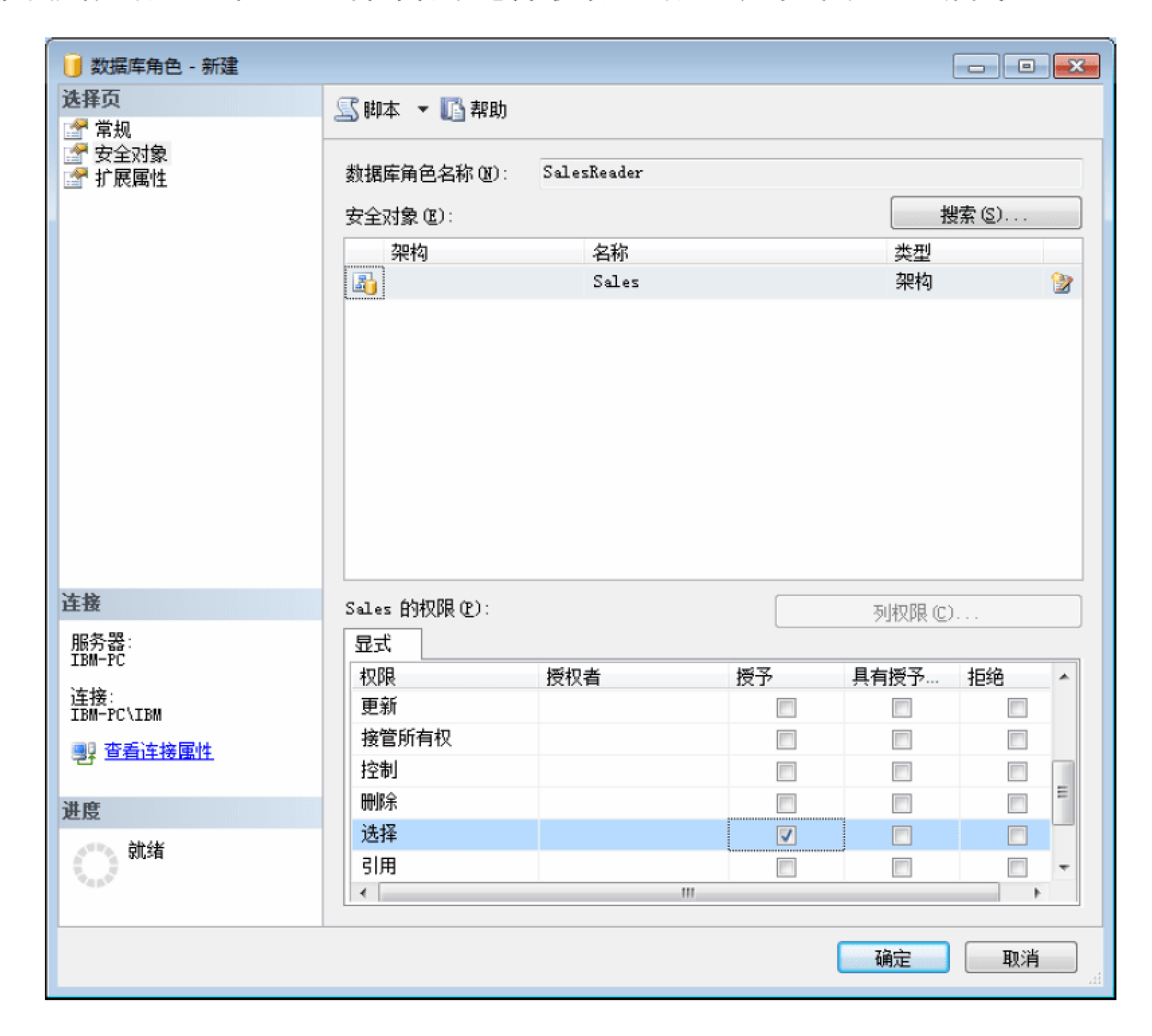


图 5.36　配置权限

（8）配置完成后单击“确定”按钮，系统将在 AdventureWorks2012 数据库中建立 SalesReader 角色，并将 test2 用户添加到该角色中。添加成功角色后可以通过对象资源管理器看到新建的角色，如图 5.37 所示。

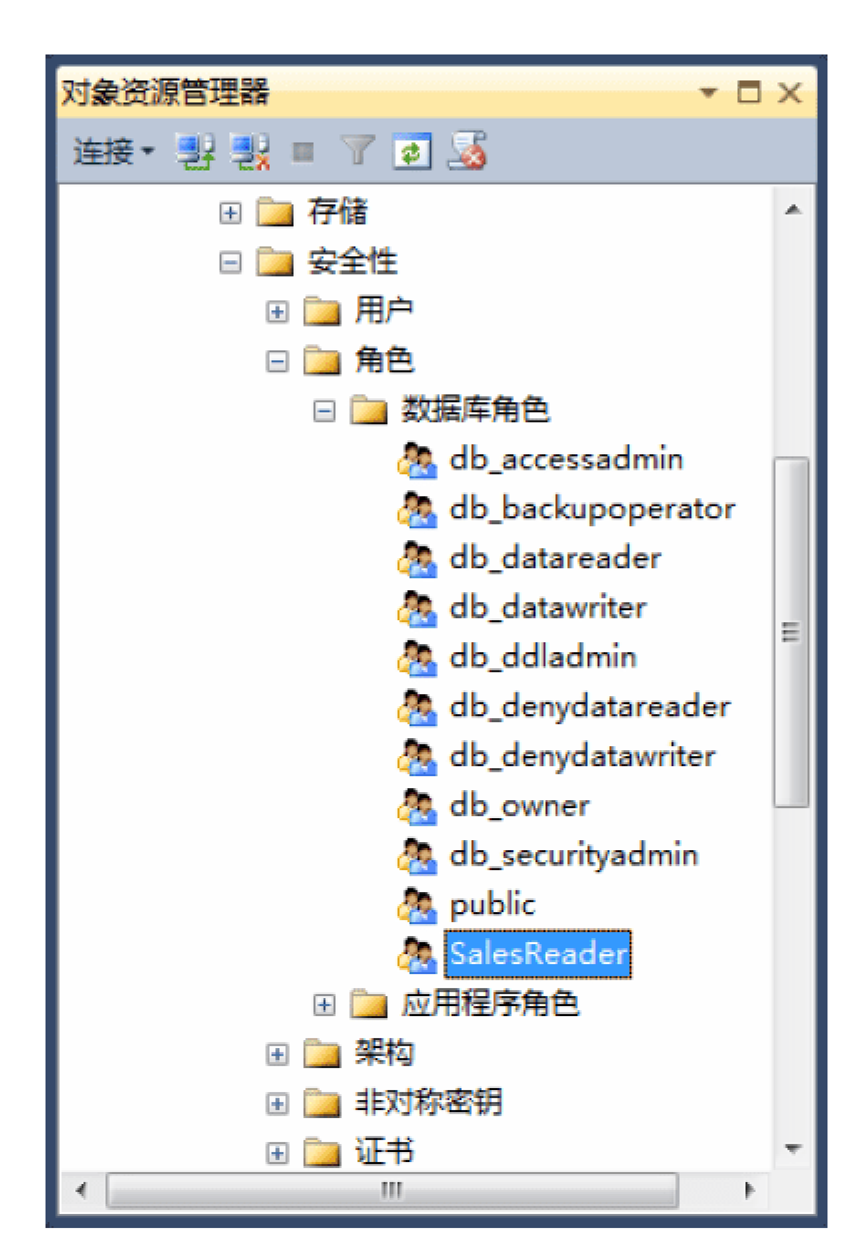


图 5.37　在对象资源管理器中查看角色

删除角色非常简单，与删除架构删除用户类似，在 SQL Server 2012 中删除角色使用 DROP ROLE 命令。例如要删除 AdventureWorks2012 数据库中创建的角色 HRReader，则对应的 SQL 脚本如代码 5.32 所示。

代码 5.32　删除角色

```
USE AdventureWorks2012
GO
DROP role HRReader
```

注意：无法从数据库删除拥有成员的角色，在要删除用户定义的数据库角色之前必须要清空该角色中的所有用户；否则将会删除角色失败。

使用 SSMS 删除角色的操作与删除用户、架构等并没有什么不同。只需要在 SSMS 的对象资源管理器中选中该角色，然后使用快捷键 Delete，系统将弹出删除对象对话框，单击“确定”按钮即可完成角色的删除。

5.7.5　应用程序角色

应用程序角色是特殊的数据库角色，用于允许用户通过特定应用程序获取特定数据。应用程序角色不包含任何成员，而且在使用它们之前要在当前连接中将它们激活。激活一个应用程序角色后，当前连接将丧失它所具备的特定用户权限，只获得应用程序角色所拥有的权限。

应用程序角色能够在不断开连接的情况下切换用户的角色和对应的权限。应用程序角色的使用过程如下：

（1）用户通过登录名或 Windows 认证方式登录到数据库。

（2）登录有效，获得用户在数据库中拥有的权限。

（3）应用程序执行 sp_setapprole 系统存储过程并提供角色名和口令。

（4）应用程序角色生效，用户原有角色对应的权限消失，用户将获得应用程序角色对应的权限。

（5）用户使用应用程序角色中的权限操作数据库。

要创建应用程序角色，需要使用 SQL Server 2012 中的 CREATE APPLICATION ROLE 命令。该命令的语法如代码 5.33 所示。

代码 5.33　创建应用程序角色语法

```
CREATE APPLICATION ROLE application_role_name
WITH PASSWORD = 'password'
[ , DEFAULT_SCHEMA = schema_name ]
```

其中，application_role_name 为指定应用程序角色的名称。该名称不能被用于引用数据库中的任何主体。PASSWORD = 'password'用于指定数据库用户将用于激活应用程序角色的密码，应始终使用强密码。DEFAULT_SCHEMA = schema_name 用于指定服务器在解析该角色的对象名时将搜索的第一个架构。如果未定义 DEFAULT_SCHEMA，则应用程序角色将使用 DBO 作为其默认架构。schema_name 可以是数据库中不存在的架构。

例如，要在 AdventureWorks2012 数据库中创建应用程序角色 PersonReader，该角色的

密码为 123，那么对应的 SQL 脚本如代码 5.34 所示。

代码 5.34　创建应用程序角色

```
USE AdventureWorks2012
GO
CREATE APPLICATION ROLE [PersonReader]
WITH DEFAULT_SCHEMA = [dbo],
PASSWORD ='123'
```

在创建好应用程序角色后接下来就是为该角色分配权限，这里仍然使用 GRANT 等权限分配命令。要将 Person 架构的表的 SELECT 权限分配给该角色的脚本如代码 5.35 所示。

代码 5.35　为应用程序角色分配权限

```
USE AdventureWorks2012;
GO
GRANT SELECT
ON SCHEMA::Person
TO PersonReader
```

应用程序角色的配置已经完成，接下来就是使用了。首先使用 testuser1 登录，该登录名在 AdventureWorks2012 数据库中对应用户 test1，该用户对 Sales 架构的表有选择权限，那么运行代码 5.36 后，系统将会抛出异常，因为没有对 Person.AddressType 表的访问权限。

代码 5.36　使用 testuser1 访问数据库

```
USE AdventureWorks2012;
GO
SELECT TOP 10 *
FROM Sales.Customer
GO
SELECT * --这里将会抛出异常，因为没有权限访问
FROM Person.AddressType
GO
```

要使用应用程序角色，需要调用系统存储过程 sp_setapprole。该存储过程的语法如代码 5.37 所示。

代码 5.37　sp_setapprole 的语法

```
sp setapprole [ @rolename = ] 'role',
       [ @password = ] { encrypt N'password' } | 'password'
       [ , [ @encrypt = ] { 'none' | 'odbc' } ]
       [ , [ @fCreateCookie = ] true | false ]
       [ , [ @cookie = ] @cookie OUTPUT ]
```

这里需要最主要的参数是 role，即要使用的应用程序角色名，password 为该应用程序角色对应的密码。其他参数都是可选参数，读者若需深入学习可以查看帮助文档。

同样使用 testuser1 登录，执行代码 5.38，系统将会抛出异常，因为当前用户的角色已经切换到应用程序角色中，用户不再对 Sales.Customer 表具有访问权限。

代码 5.38　激活应用程序角色

```
USE AdventureWorks2012;
```

```
GO
EXEC sp setapprole PersonReader,'123'  --激活应用程序角色
GO
SELECT TOP 10 *  --这里抛出异常，因为应用程序角色 PersonReader 并没有对该表的访问
                   权限
FROM Sales.Customer
GO
SELECT * --正常访问
FROM Person.AddressType
```

注意：应用程序角色是单向的，也就是说当前用户一旦切换到应用程序角色将不能再切换回原来的角色中。若需要使用原来用户的角色只有终止当前连接并重新登录。

若要删除应用程序角色，需要使用 DROP APPLICATION ROLE 命令。例如要删除前面创建的应用程序角色 PersonReader，则对应的 SQL 脚本如代码 5.39 所示。

代码 5.39　删除应用程序角色

```
USE AdventureWorks2012;
GO
DROP APPLICATION ROLE PersonReader
```

在 SSMS 中以在 AdventureWorks2012 数据库中创建应用程序角色 PersonReader 为例，在 SSMS 中创建该应用程序角色的主要操作步骤如下所述。

（1）在 SSMS 的对象资源管理器中依次展开数据库、AdventureWorks2012、安全性、角色、应用程序角色节点。

（2）右击“应用程序角色”节点，在弹出的快捷菜单中选择“新建应用程序角色”选项，系统将弹出“应用程序角色-新建”对话框，如图 5.38 所示。

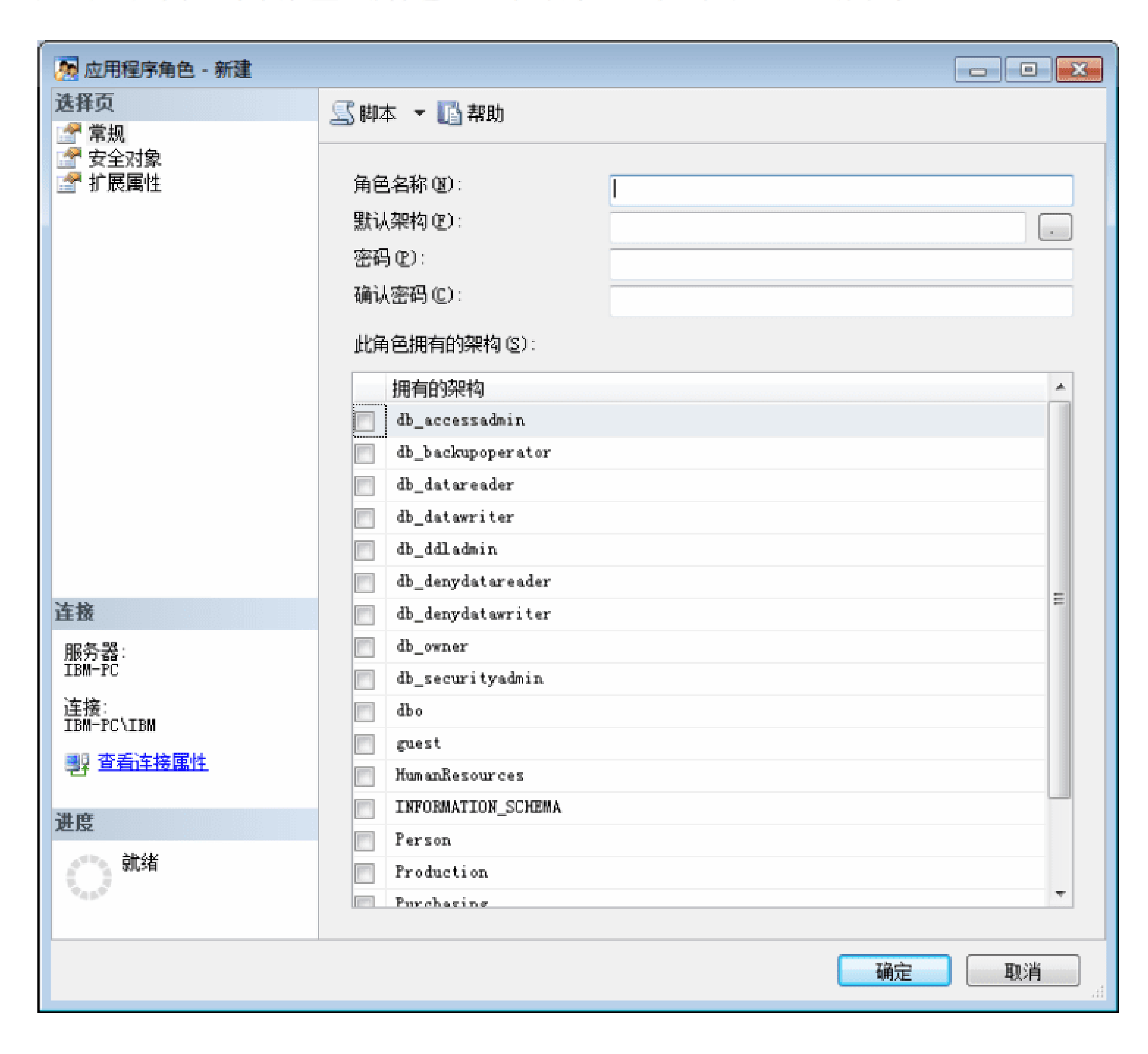


图 5.38　“应用程序角色-新建”对话框

（3）在“角色名称”文本框中输入要新建的角色名 PersonReader，在“默认架构”文本框中输入 dbo，在“密码”和“确认密码”文本框中输入角色的密码 123。

（4）选择“安全对象”选项，系统切换到应用程序角色的权限配置窗口。

（5）应用程序角色的权限配置和数据库角色的权限配置及用户的权限配置对话框相同，用同样的操作为该角色配置对 Person 架构的 SELECT 权限（读者若不清楚如何操作，可以参看前面的“使用 SSMS 管理用户权限”小节）。配置后如图 5.39 所示。

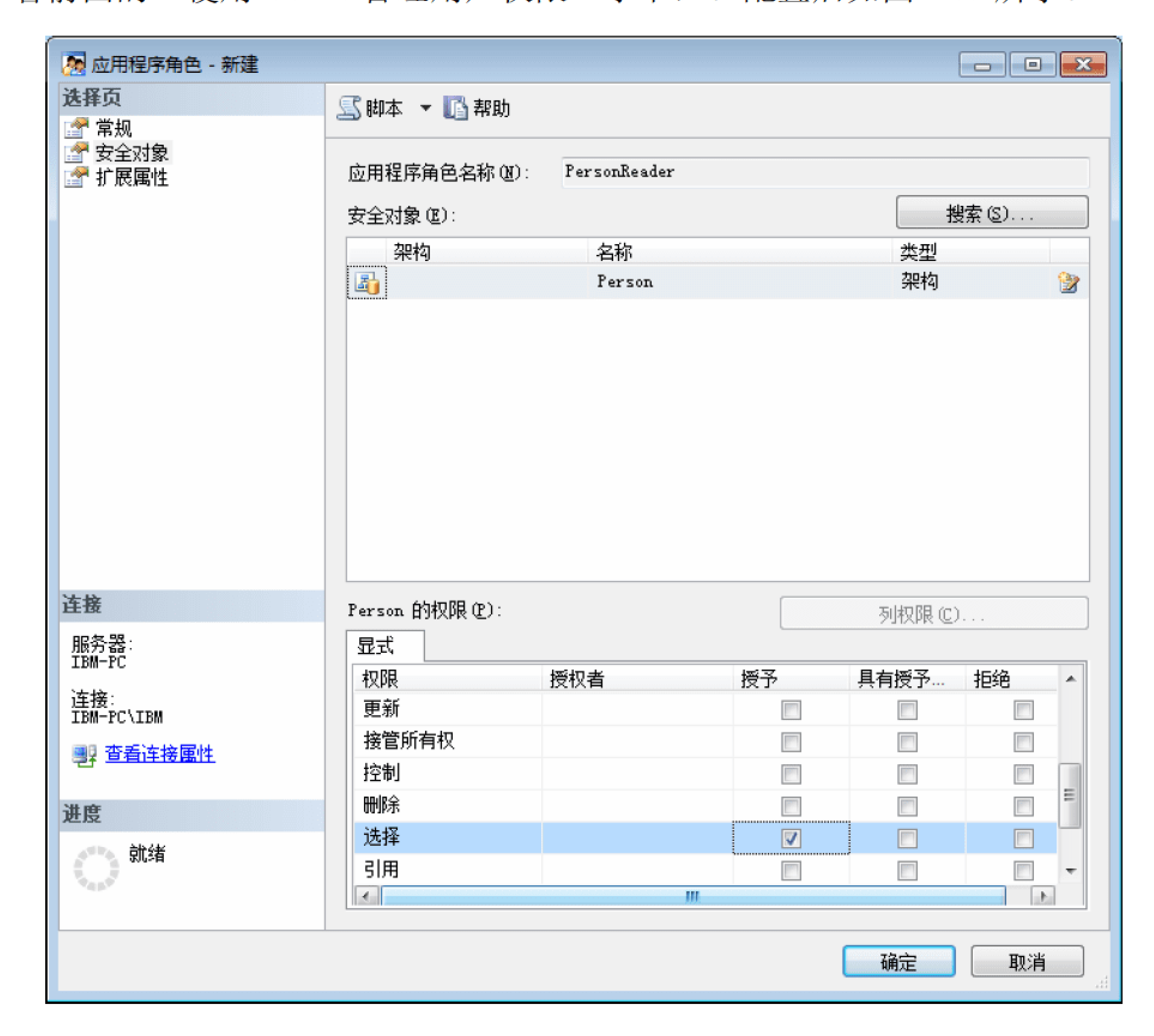


图 5.39　配置应用程序角色权限

（6）单击“确定”按钮即可完成应用程序角色的创建。

若要在 SSMS 中删除应用程序角色，其操作和删除用户数据库角色相同。在对象资源管理器中选中需要删除的应用程序角色，然后使用快捷键 Delete，在弹出的删除对象对话框中单击“确定”按钮即可实现删除操作。

5.8　数 据 加 密

在评估安全框架的过程中，企业的 IT 部门可能需要重新评估整个组织的安全性。这些安全措施可包括前面提到的密码策略、审核策略、数据库服务器隔离，以及应用程序验证和授权控制。但是，保护敏感数据的最后一个安全屏障通常是数据加密，本节将主要讲解数据加密在 SQL Server 中的应用。

5.8.1　数据加密简介

加密是一种帮助保护数据的机制。加密是通过使用特定的算法将数据打乱，达到只有经过授权的人员才能访问和读取数据的目的，从而帮助提供数据的保密性。当原始数据（称为明文）与称为密钥的值一起经过一个或多个数学公式处理后，数据就完成了加密。此过程使原始数据转为不可读形式。获得的加密数据称为密文。为使此数据重新可读，数据接收方需要使用相反的数学过程以及正确的密钥将数据解密。

然而，加密时需要执行某种算法，此过程会增加计算机处理器时间，加密后的密文一般会比明文数据大，密文的存储也需求更多的成本。较长的加密密钥比较短的加密密钥更有助于提高密文的安全性。不过，较长的加密密钥的加密/解密运算更加复杂，占用的处理器时间也比较短的加密密钥长。一般有以下两种主要加密类型：

- 对称加密。此种加密类型又称为共享密钥加密。
- 非对称加密。此种加密类型又称为两部分加密或公共密钥加密。

对称加密使用相同的密钥加密和解密数据，如图 5.40 所示。对称加密使用的算法比用于非对称加密的算法简单。由于这些算法更简单以及数据的加密和解密都使用同一个密钥，所以对称加密比非对称加密的速度要快得多。因此，对称加密适合大量数据的加密和解密。常用的对称加密算法有：RC2（128 位）、3DES 和 AES 等。

非对称加密使用两个具有数学关系的不同密钥加密和解密数据。这两个密钥分别称为私钥和公钥。它们合称为密钥对。使用密钥对进行加密解密的过程如图 5.41 所示。非对称加密被认为比对称加密更安全，因为数据的加密密钥与解密密钥不同。但是，由于非对称加密使用的算法比对称加密更复杂，并且还使用了密钥对，因此当组织使用非对称加密时，其加密过程比使用对称加密慢很多。常用的非对称加密算法有：RSA 和 DSA。



图 5.40　对称加密　　　　图 5.41　非对称加密

注意： 加密不仅对 CPU 和内存造成一定的性能影响，加密后的数据占用的存储空间也会有所改变，加密后数据大小取决于使用的算法、密钥的大小和明文的大小。

SQL Server 2012 提供了内置的数据加密功能，并支持以下 3 种加密类型，每种类型使用一种不同的密钥，并且具有多个加密算法和密钥强度。

- 对称加密：SQL Server 2012 中支持 RC4、RC2、DES 和 AES 系列加密算法。
- 非对称加密：SQL Server 2012 支持 RSA 加密算法，以及 512 位、1024 位和 2048 位的密钥强度。
- 证书：使用证书是非对称加密的另一种形式。但是，一个组织可以使用证书并通过数字签名将一组公钥和私钥与其拥有者相关联。SQL Server 2012 支持“因特网工程工作组”（IETF） X.509 版本 3（X.509v3）规范。一个组织可以对 SQL Server

2012 使用外部生成的证书，或者可以使用 SQL Server 2012 生成证书，证书可以以独立文件的形式备份，然后在 SQL Server 中进行还原。

SQL Server 2012 用分层加密和密钥管理基础结构来加密数据。每一层都使用证书、非对称密钥和对称密钥的组合对它下面的一层进行加密，顶级（服务主密钥）是用 Windows DP API 加密的。如图 5.42 所示的加密层次结构与权限层次结构中介绍的安全对象的层次结构相似。

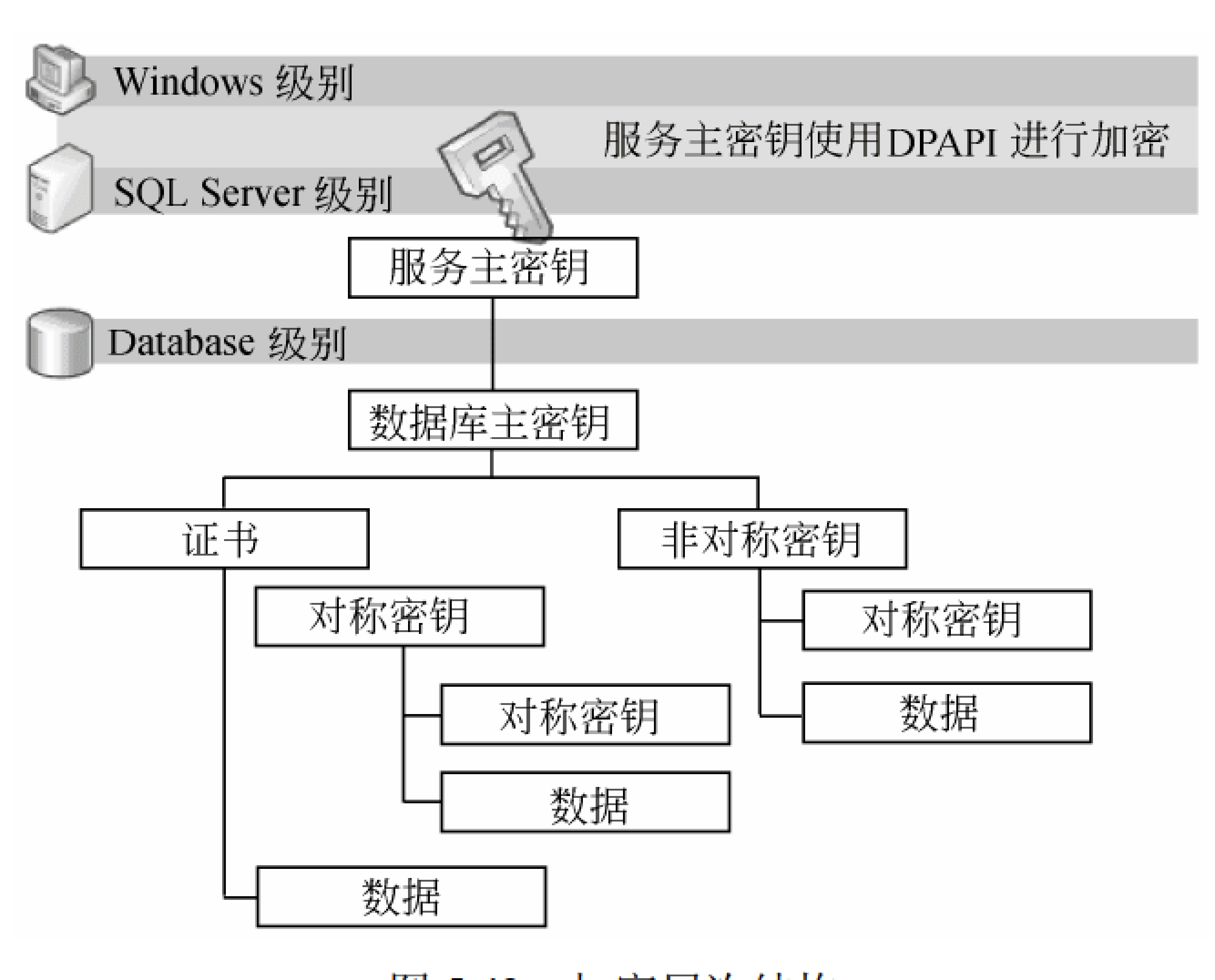


图 5.42　加密层次结构

5.8.2　数据的加密和解密

在对加密和解密有了一个基本的概念后，本小节将在 SQL Server 中对数据进行加密和解密。SQL Server 中有些数据列是十分敏感的，例如用户的密码、信用卡号、员工的工资等，这些数据如果未经过加密，一旦数据库内容泄露，将造成不可估量的损失。所以，在 SQL Server 中需要将这些数据库进行加密后保存，这样即使数据库文件被盗，别人在没有密钥的情况下是无法查看这些敏感数据的。

以一个测试数据库 TestDB1 为例，这其中有一个管理员表 AdminUser，该表保存了管理员的用户名 LoginName 和密码 Password。在 SQL Server 2012 中，要使用对称加密算法对 Password 数据列进行加密主要经过以下几步。

（1）创建数据库主密钥。数据库主密钥又叫服务主密钥，为 SQL Server 加密层次结构的根。服务主密钥是首次需要它来加密其他密钥时自动生成的。默认情况下，服务主密钥使用 Windows 数据保护 API 和本地计算机密钥进行加密。只有创建服务主密钥的 Windows 服务账户或有权访问服务账户名称和密码的主体能够打开服务主密钥。

SQL Server 中的数据库级别加密功能依赖于数据库主密钥。创建数据库时不会自动生成该密钥，必须由系统管理员创建，仅需要对每个数据库创建一次主密钥即可。SQL Server 中创建主密钥使用如下命令：

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = '<password>'
```

其中，password 为创建主密钥时对主密钥副本进行加密的密码。这里创建主密钥的脚本如代码 5.40 所示。

代码 5.40　创建主密钥

```
USE TestDB1;
GO
CREATE MASTER KEY  --创建主密钥
ENCRYPTION BY PASSWORD ='P@ssw0rd' --指定密码
```

（2）创建一个证书。SQL Server 2012 使用证书加密数据或对称密钥。公钥证书（通常只称为证书）是一个数字签名语句，它将公钥的值绑定到拥有对应私钥的人员、设备或服务的标识上。证书是由证书颁发机构（CA）颁发和签名的。从 CA 处接收证书的实体是该证书的主体。证书中通常包含下列信息：

- 主题的公钥。
- 主题的标识符信息，如姓名和电子邮件地址。
- 有效期。这是指证书被认为有效的时间长度。
- 颁发者标识符信息。
- 颁发者的数字签名。

说明： 证书只有在指定的有效期内有效，每个证书都包含一个“有效期始于”和“有效期至”日期。这两个日期设置了有效期的界限。证书超过有效期后，必须由已过期证书的主题请求一个新证书。

SQL Server 提供了 CREATE CERTIFICATE 命令用于创建证书。这里为加密创建证书的脚本如代码 5.41 所示。

代码 5.41　创建证书

```
USE TestDB1;
GO
CREATE CERTIFICATE AdminPwdCert
   WITH SUBJECT = 'TO Encrypt Admin Password', --证书的主题
   EXPIRY_DATE = '2013/1/1';--证书的过期日期
```

（3）创建一个对称密钥，以加密目标数据。使用第（2）步中创建的证书、其他对称密钥或用户提供的密码加密此对称密钥。 SQL Server 提供了 CREATE SYMMETRIC KEY 命令用于创建对称密钥。此处使用 AES 256 加密算法用于创建密钥，则对应的脚本如代码 5.42 所示。

代码 5.42　创建对称密钥

```
USE TestDB1;
GO
CREATE SYMMETRIC KEY PwdKey
WITH ALGORITHM = AES_256 --使用 AES 256 加密算法
ENCRYPTION BY CERTIFICATE AdminPwdCert;--使用证书加密
```

（4）打开对称密钥将数据加密或解密。要打开此密钥，可以使用以下命令：

```
OPEN SYMMETRIC KEY Key_name
   DECRYPTION BY CERTIFICATE certificate_name
```

其中 Key_name 为要打开的对称密钥的名称。certificate_name 为证书的名称，该证书的私钥将用于解密对称密钥。这里要打开前面创建密钥 PwdKey 的脚本如代码 5.43 所示。

代码 5.43　打开对称密钥

```
OPEN SYMMETRIC KEY PwdKey
   DECRYPTION BY CERTIFICATE AdminPwdCert
```

注意：打开的对称密钥将绑定到会话而不是安全上下文。 打开的密钥将持续有效，直到它显式关闭或会话终止。

（5）使用 EncryptByKey()函数加密数据，或使用 DecryptByKey()函数解密数据。至此，该数据在数据库中存储为二进制大对象（BLOB）或者被解密，这取决于使用的 Transact-SQL 语句。加密函数 EncryptByKey()的语法格式为：

```
EncryptByKey (key_GUID , 'cleartext' )
```

其中 key_GUID 为密钥的 GUID 值，可以通过 Key_GUID('key_name')函数获得该值。第二个参数 cleartext 就是要加密的明文。例如要插入加密密码的管理员数据操作如代码 5.44 所示。

代码 5.44　插入加密数据

```
CREATE TABLE AdminUser
(
    LoginName varchar(50) NOT NULL PRIMARY KEY,
    Password varbinary(500) NOT NULL
)
GO
INSERT INTO AdminUser
VALUES('admin1',EncryptByKey(Key_GUID('PwdKey'), 'p@ssw0rd1')) --加密数据
```

解密函数 DecryptByKey()只需传入密文，该函数将会返回解密出的明文。

注意：DecryptByKey()函数返回的是 varbinary 数据，需要经过数据类型转换才能阅读。DecryptByKey()使用对称密钥，该对称密钥必须已经在数据库中打开，可以同时打开多个密钥。不必只在解密密码之前才打开密钥。

解密并查询数据的脚本如代码 5.45 所示。

代码 5.45　解密数据

```
SELECT LoginName,Password --直接查询的内容将不可读
FROM AdminUser
SELECT LoginName,CONVERT(varchar(50),DecryptByKey(Password)) –解密出明文
FROM AdminUser
```

（6）关闭对称密钥。关闭对称密钥使用 CLOSE SYMMETRIC KEY 命令。该命令的语法为：

```
CLOSE { SYMMETRIC KEY key_name | ALL SYMMETRIC KEYS }
```

其中 CLOSE SYMMETRIC KEY key_name 为关闭指定的密钥，而 CLOSE ALL SYMMETRIC KEYS 为关闭所有打开的密钥。这里关闭对称密钥的脚本为：

```
CLOSE SYMMETRIC KEY PwdKey
```

注意：关闭密钥后加密函数 EncryptByKey()和解密函数 DecryptByKey()都将无效，必须重新打开密钥才能使用。

5.8.3　使用证书加密和解密

通常情况下，使用对称密钥加密数据，此方法利用了对称加密速度快的优点。但是也可以使用证书代替对称密钥将数据加密。由于非对称加密比对称加密更安全，因此，当需要在运行 SQL Server 2012 的多台服务器间传输加密密钥的情况下，使用证书加密数据很有用。使用证书进行加密和解密主要经过以下几步操作。

（1）创建数据库主密钥。具体创建操作在 5.8.2 节已经做了介绍，这里不再重复介绍。一个数据库中只有一个主密钥，如果已经创建过主密钥就不再重复创建了。

（2）创建一个证书。具体操作也与 5.8.2 节介绍的相同。这里假设需要将员工的工资字段进行加密，创建一个新的证书用于加密，创建脚本如代码 5.46 所示。

代码 5.46　创建证书

```
USE TestDB1;
GO
CREATE CERTIFICATE WageCert
   WITH SUBJECT = 'TO Encrypt Wage',      --证书的主题
   EXPIRY_DATE = '2013/12/31';            --证书的过期日期
```

（3）使用证书的公钥加密数据。使用证书加密数据需要用到 EncryptByCert()函数。该函数返回 varbinary 类型数据，其语法为：

```
EncryptByCert ( certificate_ID , { 'ciphertext' | @ciphertext })
```

其中，certificate_ID 为证书的 ID，可以通过 Cert_ID('cert_name')函数获得证书 ID。'cleartext'为要进行加密的明文。使用证书加密工资字段的脚本如代码 5.47 所示。

代码 5.47　使用证书加密数据

```
CREATE TABLE Employee
(
    EmpID int NOT NULL PRIMARY KEY,
    Wage varbinary(500) NOT NULL --工资字段，加密后为二进制数据
)
GO
INSERT INTO Employee
VALUES(1, EncryptByCert(Cert_ID('WageCert'),'5000')) --使用证书加密
```

（4）使用证书的私钥解密数据。使用证书解密数据需要用到 DecryptByCert()函数。该函数返回 varbinary 类型数据，其语法为：

```
DecryptByCert (certificate_ID , { 'ciphertext' | @ciphertext })
```

其中，certificate_ID 为证书的 ID，'ciphertext'为经过加密后的密文。使用证书解密工资字段的脚本如代码 5.48 所示。

代码 5.48　使用证书解密数据

```
SELECT *  --直接查询数据，Wage 字段是加密的
FROM Employee
SELECT EmpID,CONVERT(varchar(50),
DECRYPTBYCERT(Cert_ID('WageCert'),Wage))  --使用证书解密 Wage 字段
FROM Employee
```

注意：使用证书加密是非对称加密操作，将会消耗大量资源，所以不提倡在常用的数据列上使用。

5.8.4　使用透明数据加密

透明数据加密旨在为整个数据库提供静态保护而不影响现有的应用程序。透明数据加密可对数据和日志文件进行实时的 I/O 加密和解密。这种加密使用数据库加密密钥（DEK），该密钥存储在数据库启动记录中以供恢复时使用。DEK 通过存储在服务器的 master 数据库中的证书来保证安全。

数据库文件的加密在页级执行。已加密数据库中的页在写入磁盘之前会进行加密，在读入内存时会进行解密。透明数据加密不会增大已加密数据库的大小。以对 TestDB1 数据库使用 TDE 为例，主要操作步骤如下所述。

（1）创建数据库主密钥。创建数据库主密钥使用 CREATE MASTER KEY 命令，前面已经做了介绍，创建脚本如代码 5.49 所示。

代码 5.49　创建数据库主密钥

```
USE master;
GO
CREATE MASTER KEY --创建数据库主密钥
    ENCRYPTION BY PASSWORD = 'password';
```

注意：使用 TDE 时创建的数据库主密钥是在 master 系统数据库中创建的，而前面提到的对称加密和证书加密都是在具体的目标数据库中创建的。

（2）创建一个证书。创建证书使用 CREATE CERTIFICATE 命令，关于证书的创建在前面内容中已经做了介绍。创建证书的脚本如代码 5.50 所示。

代码 5.50　创建证书

```
USE master;
GO
CREATE CERTIFICATE --创建证书
tdeCert WITH SUBJECT = 'use to TDE';
```

说明： 在不指定 EXPIRY_DATE 参数的情况下，证书默认为从当前时刻生效，1 年后失效。

（3）创建用于以透明方式加密数据库的加密密钥。在 master 数据库创建好用于 TDE 的证书后，接下来就需要使用 SQL Server 提供的 CREATE DATABASE ENCRYPTION KEY 命令在需要被加密的数据库中创建加密密钥。该命令的语法如代码 5.51 所示。

代码 5.51　CREATE DATABASE ENCRYPTION KEY 命令的语法

```
CREATE DATABASE ENCRYPTION KEY
       WITH ALGORITHM = { AES_128 | AES_192 | AES_256 | TRIPLE_DES_3KEY }
   ENCRYPTION BY SERVER CERTIFICATE Encryptor_Name
```

其中，AES_128、AES_192、AES_256、TRIPLE_DES_3KEY 都是用于指定加密密钥的加密算法。Encryptor_Name 指定用于加密数据库密钥的加密程序名称，即证书的名称。

假设现在需要对 TestDB1 数据库使用 AES_256 加密算法进行透明数据加密，则在该数据库上创建加密密钥的脚本如代码 5.52 所示。

代码 5.52　创建加密密钥

```
USE TestDB1;
GO
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_256 --指定加密算法
ENCRYPTION BY SERVER CERTIFICATE tdeCert;
```

（4）修改数据库，使 TDE 可用。创建好加密密钥后数据库并没有进行加密，必须要修改数据库，开启加密选项，使 TDE 可用。开启 TDE 后，系统将在后台开启一个进程进行异步的加密扫描，直到将现有数据库中的所有数据加密完成。代码 5.53 用于修改数据库开启 TDE 加密。

代码 5.53　修改数据库开启 TDE

```
ALTER DATABASE TestDB1
SET ENCRYPTION ON   --修改数据库，开启透明数据加密
```

TDE 之所以被称为透明数据加密，是因为它只是对数据库的数据文件和日志文件进行加密，对用户和程序而言并不会有任何改变，也就是说，这个加密操作对用户来说是透明的。用户对数据库的写操作都会由系统将数据加密后再写到数据文件和日志文件上，同样，读操作也是先将加密的数据读取出来由系统解密后再返回给用户。

5.9　SQL 注入攻击

在数据库应用开发中，有时由于程序员的水平及经验不足，在编写代码时，没有对用户输入数据的合法性进行判断，使应用程序存在安全隐患。用户可以提交一段数据库查询代码，根据程序返回的结果获得某些想得知的数据，这就是所谓的 SQL Injection，即 SQL 注入。

5.9.1　SQL 注入攻击原理

SQL 注入是由于未对用户输入的数据进行合法性判断造成的。为了便于读者理解，这里就以一个新闻系统为例，现有一个新闻展示页面 News.aspx，该页面根据 URL 中跟的参数 id 来决定读取哪一条新闻。如果为对 URL 中的参数进行判断，采用拼 SQL 语法的方式读取新闻数据的程序段如代码 5.54 所示。

代码 5.54　读取新闻数据

```
protected void Page_Load(object sender, EventArgs e)
{
    string sql = "SELECT * FROM News WHERE NewsID="
    + Request.QueryString["id"];//这里就是 URL 中传入的 id 参数
    BindNews(sql); //将 SQL 语句传入，根据 SQL 语句读取信息
}
```

虽然是使用 C#编写，但相信这段代码读者很容易理解。在正常访问新闻页面时，例如 http://xxxxxx/News.aspx?id=123，那么后台生成的 SQL 查询语句为：

```
SELECT * FROM News WHERE NewsID=123
```

整个语句和逻辑都没有问题，新闻数据被查出并显示在页面上。那么如果用户在 URL 后跟了其他信息呢？例如将 URL 写成 http://xxxxxx/News.aspx?id=123 and 1=1，那么后台生成的 SQL 语句为：

```
SELECT * FROM News WHERE NewsID=123 and 1=1
```

这个 SQL 语句也没有问题，数据被正常查出并绑定到页面上。这时再将参数改为“?id=123 and 1=2”，那么生成的 SQL 语句为：

```
SELECT * FROM News WHERE NewsID=123 and 1=2
```

显然这样是查不出数据的，页面上显示数据不存在或抛出异常。通过跟不同的参数，黑客就可以定位这个地方就是 SQL 注入点了。

既然发现了注入点，那么黑客又能做什么？如果当前读取新闻的用户具有超级管理员权限，那么黑客利用这个注入点，基本上什么都可以做。这里笔者举一个简单的参数情况，如果参数改为“?id=123;drop table News”，那么后台生成的 SQL 语句就变成了：

```
SELECT * FROM News WHERE NewsID=123;DROP TABLE News
```

系统运行该 SQL，整个新闻表都被删除了。另外，利用 SQL 注入漏洞还可以绕过用户认证，例如用户登录时的后台程序如代码 5.55 所示。

代码 5.55　登录验证代码

```
string sql="SELECT * FROM AdminUser WHERE LoginName='"
+txbloginName.Text                    //用户输入的用户名
+"' AND Password='"+txbPwd.Text+"';" //输入的密码
Validate(sql);                        //根据是否返回数据行来验证用户名密码是否正确
```

对于正常的用户登录，那么生成的 SQL 语句为：

```
SELECT * FROM AdminUser
WHERE LoginName='admin' AND Password='p@ssw0rd'
```

验证用户成功，用户成功登录。那么如果在用户名中填写为“admin’ --”而密码随便填写 123，那么后台生成的 SQL 语句如代码 5.56 所示。

代码 5.56　生成被注入的 SQL 代码

```
SELECT *
FROM AdminUser
WHERE LoginName='admin'
--'AND Password='123'
```

后面的密码部分 AND 语句被注释了，只需要通过登录名就可以成功登录。

SQL 注入的破坏还不仅于此，利用 SQL 注入，黑客还可以上传木马、提升权限、获得数据库所有数据，甚至还可以获得登录数据库服务器的管理员权限。

说明： 笔者在这里只是简单地讲解一下 SQL 注入的原理，旨在提高读者的安全意识，读者若对 SQL 注入有兴趣可自行研究。SQL 注入攻击是一种黑客行为，读者可以出于学习的目的在自己机器上测试，切不可对互联网上的网站进行攻击破坏。

5.9.2　如何防范 SQL 注入攻击

既然了解了 SQL 注入的原理，那么就可以使用对应的办法进行防范。防范 SQL 注入攻击最推荐的办法就是使用存储过程。存储过程中将使用参数来传递用户的输入，如对应查询新闻的存储过程如代码 5.57 所示。

代码 5.57　查询新闻的存储过程

```
CREATE PROC GetNewsByNewsID
@newsID int
AS
SELECT *
FROM News
WHERE NewsID=@newsID
```

由于此处定义了传入的参数必须是整数，所以“123 and 1=1”等这样的参数是无法传入存储过程的，自然也就无法运行注入的代码。对于字符串的情况也是一样的，将验证用户的数据库操作写为存储过程，对应脚本如代码 5.58 所示。

代码 5.58　验证用户的存储过程

```
CREATE PROC GetAdminByLoginNameAndPassword
@loginName varchar(50),
@password varchar(50)
AS
SELECT *
FROM AdminUser
WHERE LoginName=@loginName AND Password=@password
```

当用户再在用户名中输入“admin’ –”时，整个输入将作为一个字符串参数传入数据库，由于将整个输入作为字符串处理，所以 WHERE 条件最终变为：

```
WHERE LoginName='admin'' --' AND Password='123'
```

这样注入对存储过程就无效了。存储过程能够防止大部分 SQL 注入的发生，但并不是全部。如果用户在存储过程中动态拼接 SQL 语句，然后使用 EXEC 命令来执行动态 SQL 语句仍然会造成 SQL 注入攻击。

防范 SQL 注入的另外一种办法就是将关键字过滤或替换掉。例如将“’”符号全部替换为“””符号。如果用户输入中包含有“--”字符串，由于该字符串在 SQL 语句中表示注释，可以使用程序将该字符串替换成空字符串。另外还有些敏感的 SQL 关键字也可以列入过滤字符串中。使用字符串过滤后即使在存储过程中动态执行拼写的 SQL 语句也不会造成注入漏洞。

使用存储过程和字符串过滤的方式就可以防范 SQL 注入攻击，为了提高用户体验和系统安全性，还可以在客户端做输入合法性检查、限制用户输入长度等。另外还应该对应用程序使用的账号做严格的权限管理，不要随便将超级管理员账号给应用程序使用。

5.10　小　　结

本章主要讲解了 SQL Server 2012 在数据库安全上的一些特性和相关知识。主要包括登录名的管理、用户的管理、架构管理、用户权限设置、角色管理、数据加密和 SQL 注入等。

SQL Server 2012 通过用户角色权限的方式来实现数据库的权限管理。用户的权限是指是否允许在数据库特定对象上执行特定行为，由于数据库对象众多，而且执行的行为种类也很多，所以使用角色来管理用户权限。角色分为服务器角色和数据库角色，服务器角色是固定的不可被用户创建的，数据库角色则可以由用户创建。在创建角色后可以对角色设置用户权限，然后再对用户设置角色，从而实现了对数据库用户权限的配置。

为了保护敏感数据不被非法获取，SQL Server 2012 中支持使用密钥和证书对特定字段进行加密和解密。此外，还可以使用透明数据加密功能，实现对整个数据库的数据文件和日志文件的加密和解密。

第 6 章　数据文件安全与灾难恢复

随着信息技术的发展和计算机的普及，越来越多的数据以比特的形式保存到数据库中，数据文件作为数据的载体其安全性受到了极大重视。在发生火灾、人为操作失误、黑客入侵破坏和服务器故障等灾难时，通过什么措施来保证数据文件的安全和灾难恢复就是本章将要讲解的主要内容。

6.1　数据文件安全简介

数据库的安全不仅仅需要通过权限设置、加密等方式来保证，更需要保证数据文件不被损坏，不丢失。本节将主要对数据文件安全进行简单介绍。

6.1.1　业务可持续性

业务可持续性是指业务系统的核心功能不受外界影响，即使在灾难发生后仍然可以持续运行。为了使业务系统具有更高的可持续性，需要有相应的业务可持续性计划。业务可持续计划是一种先知先觉流程，确认影响业务关键因素及其可能面临的威胁，拟订一系列计划与步骤，以确保处于任何状况下，这些关键因素都能正常而持续发挥业务作用。业务可持续性计划包括灾备计划和对相关人、流程及技术的管理。

对业务可持续性来说，一个相当重要的评判标准就是高可用性。高可用性用系统资源的被使用时间百分比来表示，其计算公式为：

```
系统资源可被使用的时间百分比=（总体时间-不可用时间）/总体时间
```

人们常用“多少个 9 的系统可用性”来表示系统的可用性情况。9 的个数越多，系统的可用性就越高，为此付出的代价也越大。如表 6.1 列出了 1～5 个 9 的可用性时间。

表 6.1　高可用性时间

多少个9	可　用　性	1年内不可用时间
1	0.989	3天，18小时，20分钟
2	0.99	3天，15小时，36分钟
3	0.999	8小时，46分钟
4	0.9999	53分钟
5	0.99999	5分钟

影响系统高可用性的主要因素是系统出现宕机情况。系统出现宕机主要分为非计划性的和计划性的两种。非计划性的宕机包括服务器故障和数据失效两种情况。服务器故障是指发生在服务器上的硬件故障（如 CPU 烧毁、主板故障等）和软件故障（如病毒爆发、驱动错误等）。而数据失效是指数据库文件事故，具体包括：

- 存储故障；
- 人为失误；
- 损毁；
- 站点事故。

计划性的宕机主要是由于系统配置更改（如参数调整、安装补丁、系统软硬件升级等）和数据更改造成。

6.1.2　SQL Server 2012 高可用性技术

为了提高业务可持续性，作为业务系统的核心，数据库中出现了多种容灾技术。数据库容灾技术考虑的因素如下：

- 故障转移时间。故障转移时间越短，投入的成本就越高。
- 自动或手动检测切换。一般情况下自动切换方式比手动切换方式投入成本高。
- 是否容忍丢失数据。容忍的数据丢失越少，投入的成本就越高。
- 粒度：实例，数据库，数据表，数据行。粒度越大，则投入的成本也越大。
- 冗余系统成本，是否需要额外软、硬件。
- 复杂度。针对现有环境和技术力量考虑复杂度。
- 是否对客户端透明。不对客户端透明则需要修改客户端代码。

SQL Server 2012 在灾备恢复上有着不同的策略，在高可用技术上，按照数据备份的方式分类，分为 3 种技术。

- 冷备技术。特点是无故障转移，在发生系统故障时可能造成数据丢失。冷备主要是做数据库的备份与恢复以及数据文件的转移。
- 温备技术。特点是手动的故障转移，在发生系统故障时可能造成数据丢失。在 SQL Server 上的温备技术主要有事务性复制、日志传送和数据库镜像——高性能模式。
- 热备技术。特点是自动的故障转移，无数据丢失。在 SQL Server 中的技术实现有数据库镜像——高可用模式和故障转移群集。

除了灾备恢复以外，SQL Server 2012 还对人为失误做出了相应的功能，那就是数据库快照技术。在 SQL Server 2000 及以前的数据库版本中，若是由于用户、应用程序或者数据库管理员操作失误造成了数据的破坏，就只有通过从数据库的备份中才能恢复，但是备份也可能不是最新的数据，从而造成数据丢失。如果使用日志传送的方式，则又会有一定的数据延迟，而使用数据库快照可以快速恢复人为失误的破坏。

另外，在数据库的管理与维护上，SQL Server 2012 也提供了在线索引操作、数据库快速恢复、数据分区、数据压缩等。如图 6.1 所示为 SQL Server 2012 在高可用性上对应的

技术。

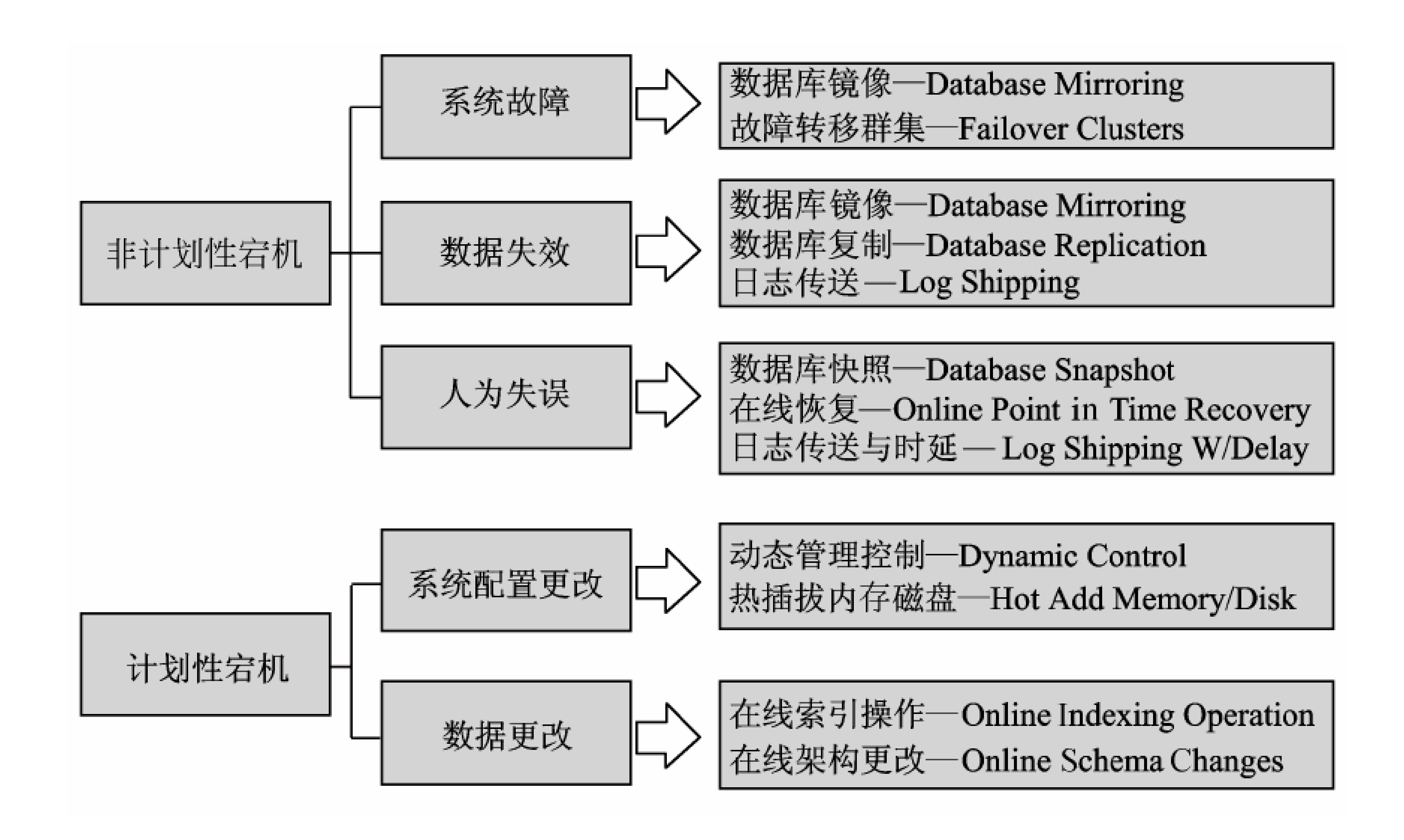


图 6.1　SQL Server 2012 高可用性

6.2　数据库的备份与恢复

数据库的备份与恢复是数据库文件管理中最常见的操作，是最简单的数据恢复方式。数据库备份在灾难恢复中起着重要的作用。本节将主要介绍数据库的备份与恢复操作。

6.2.1　数据库备份简介

对数据库的备份是最基本的一种数据库管理。在考虑备份时可以采用一个简单的规则——尽早而且经常备份。采用这一规则不是只在相同的磁盘上备份文件并遗忘它。数据库备份的文件应该存在于一个独立的远离现场的位置，以确保其安全。

备份能够提供针对数据的意外或恶意修改、应用程序错误、自然灾害的解决方案。如果以牺牲容错性为代价选择尽可能快速的数据文件访问方式，那么备份就能为防止数据损坏而提供保障。

数据库备份可以在线上环境中运行，所以根本不需要数据库离线。使用数据库备份能够将数据恢复到备份时的那一时刻，但是对备份以后的更改，在数据库文件和日志文件损坏的情况下将无法找回，这是数据库备份的主要缺点。SQL Server 2012 中提供了以下几种主要的备份类型。

- 完整备份：是将数据库中的所有页复制到另外一个备份设备上。
- 增量备份：是只复制自上次完整备份以后发生修改的区（extent，1 个区中有 8 个页）。这些修改的区会被复制到一个指定的备份设备上。SQL Server 是通过校验数据库中每个数据文件的“DCM 页”（增量变化映射页）中的比特位来分辨哪些区需要备份到设备上。

说明：DCM 页是一个很大的位图，用一个比特位来代表文件中的一个区。每次执行完整备份后 DCM 中的所有比特位清零。当一个区中的 8 个数据页有任何一个修改时，其对应的 DCM 中的比特位就会置 1。

- 日志备份：在大多数情况下，日志备份会复制自上次完整备份或日志备份后被写入的日志记录。
- 文件和文件组备份：SQL Server 中使用了文件和文件组的概念，将数据存放在多个文件组中。相对于完整备份，文件和文件组备份时只需要备份指定的某个文件和文件组，而不用像完整备份一样将整个数据库备份下来。文件和文件组备份尤其适用于大型数据库中，但前提是数据库分为了多个文件和文件组。

正是由于备份和还原数据的重要性，因此可靠的备份和还原数据需要一个备份和还原策略。设计良好的备份和还原策略可以尽量提高数据的可用性及尽量减少数据丢失。

备份和还原策略包含备份部分和还原部分。

- 备份策略：包括定义备份类型和频率、备份所需硬件和环境、测试备份的方法以及存储备份媒体的位置和方法。
- 还原策略：包括负责执行还原的人员以及执行还原来满足数据库可用性和尽量减少数据丢失的方法。

良好的备份和还原策略也需要相当的软硬件支持，所以备份还原策略应当根据实际的技术和财力之间进行权衡。

6.2.2　备份设备

SQL Server 中并不关心所有数据是备份到物理硬盘上还是磁带上。在数据库备份中，预定义的目标位置叫做设备。这里设备是对硬盘、磁带机等备份存储的统称。可以使用 T-SQL、SSMS、DMO 和 WMI 来创建备份设备。在 SQL Server 2012 中，系统提供了 sp_addumpdevice 系统存储过程用于创建备份设备。sp_addumpdevice 的语法如代码 6.1 所示。

代码 6.1　sp_addumpdevice 语法

```
sp_addumpdevice [ @devtype = ] 'device_type'
       , [ @logicalname = ] 'logical_name'
       , [ @physicalname = ] 'physical_name'
     [ , { [ @cntrltype = ] controller_type |
          [ @devstatus = ] 'device_status' }
     ]
```

其中，device_type 是备份设备的类型，备份设备类型为下列类型之一。

- Disk：本地硬盘驱动器。
- Tape：操作系统支持的任何磁带设备。

logical_name 是在 BACKUP 和 RESTORE 语句中使用的备份设备的逻辑名称。physical_name 是备份设备的物理名称。物理名称必须遵从操作系统文件名规则或网络设备的通用命名约定，并且必须包含完整路径。例如需要将数据库备份到硬盘中的“D:\DbBackup\Db1.bak”文件中，那么添加该设备的 SQL 脚本如代码 6.2 所示。

代码 6.2　添加硬盘作为设备

```
USE master;
GO
EXEC sp_addumpdevice 'disk', 'mydisckbackup', 'D:\DbBackup\Db1.bak';
```

若需要将数据库备份到网络共享硬盘上，那么对应的 SQL 脚本如代码 6.3 所示。

代码 6.3　添加网络共享硬盘作为设备

```
USE master;
GO
EXEC sp_addumpdevice 'disk', 'mynetbackup', '\\192.168.1.2\ShareBak\
Db1.bak';
```

说明： 要使用网络共享硬盘作为设备，则必须具有对该路径的读写权限。

若备份设备不再使用，需要从数据库中删除，则需要使用系统提供的 sp_dropdevice 存储过程。sp_dropdevice 的语法为：

```
sp_dropdevice [ @logicalname = ] 'device'
    [ , [ @delfile = ] 'delfile' ]
```

其中 device 为需要删除设备的逻辑名称。第二个参数 delfile 为可选参数，表示是否删除物理文件，如果是 DELETE 则表示删除。例如要删除前面创建的网络共享硬盘设备 mynetbackup，则对应的 SQL 脚本如代码 6.4 所示。

代码 6.4　删除备份设备

```
USE master;
GO
EXEC sp_dropdevice 'mynetbackup';
```

在 SSMS 中添加备份设置的主要操作如下所述。

（1）在 SSMS 的对象资源管理器中展开“服务器对象”节点下的“备份设备”节点，该节点下列出了当前系统的所有备份设备。

（2）右击“备份设备”节点，在弹出的快捷菜单中选择“新建备份设备”选项，系统将弹出“备份设备”对话框，如图 6.2 所示。

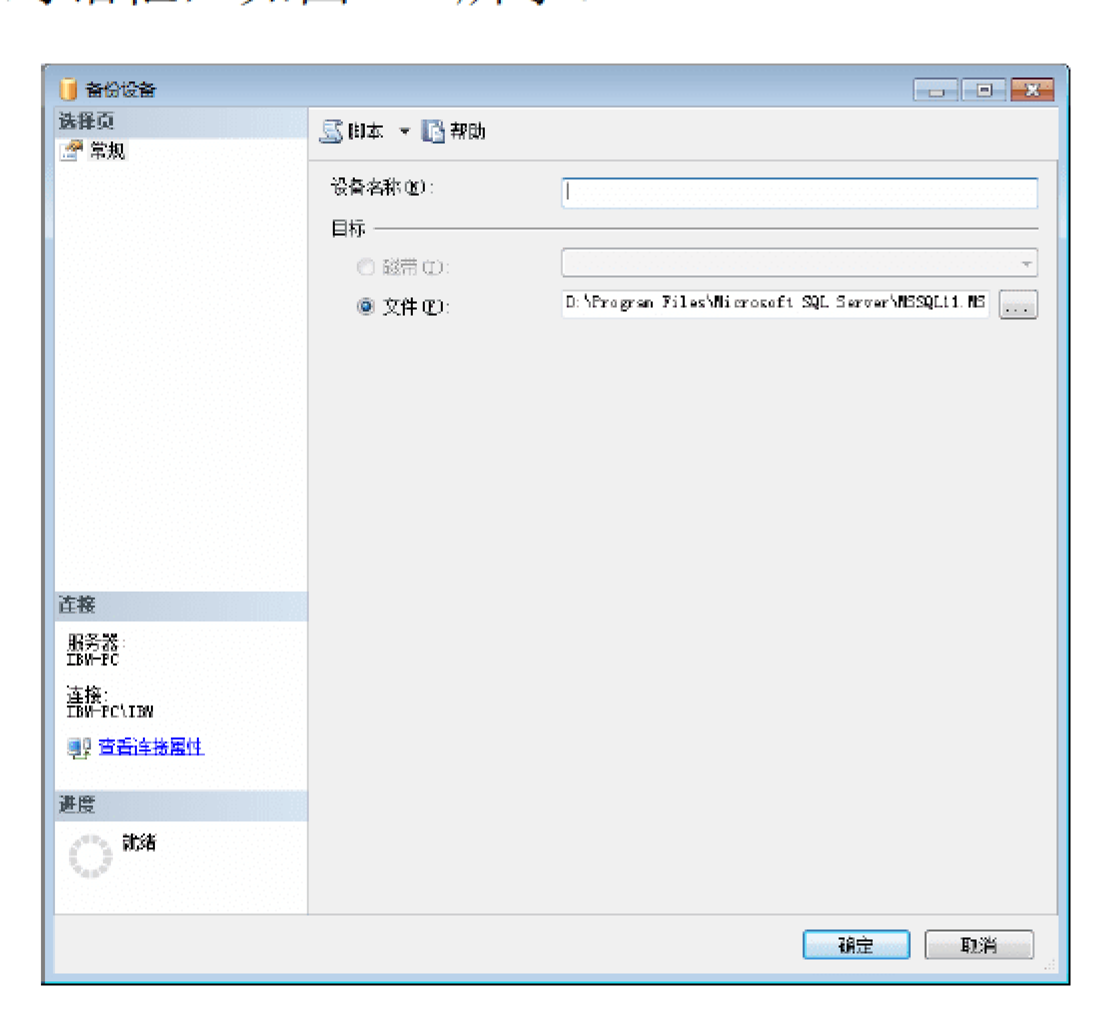


图 6.2 “备份设备”对话框

（3）在“设备名称”文本框中输入要新建的设备名称，由于笔者当前环境没有磁带机，所以“磁带”复选框是不可选中的。在“文件”文本框中选择或输入备份的硬盘路径，然后单击“确定”按钮，系统将完成设备的创建工作。

在 SSMS 中删除设备仍然是通过 Delete 快捷键来完成。

注意：备份设备只能新建和删除，不能修改。若要修改某个备份设备，只有先删除该设备，然后新建一个同样逻辑名的备份设备。

6.2.3　数据库备份

前面已经讲到，SQL Server 2012 提供了多种备份类型。SQL Server 为数据库备份提供了 BACKUP 命令，无论是完整备份、增量备份、日志备份还是文件组备份都是使用该命令。BACKUP 命令的语法格式如代码 6.5 所示。

代码 6.5　BACKUP 语法

```
BACKUP DATABASE|LOG { database_name | @database_name_var }
[<file_or_filegroup>   [   ,...n   ]   |   READ_WRITE_FILEGROUPS   [   ,
<read_only_filegroup> [ ,...n ] ] ]
TO <backup_device> [ ,...n ]
[ WITH { DIFFERENTIAL | <general_WITH_options> [ ,...n ] } ]
<general_WITH_options> [ ,...n ]::=
  COPY_ONLY
 | { COMPRESSION | NO_COMPRESSION }
 | DESCRIPTION = { 'text' | @text_variable }
 | NAME = { backup_set_name | @backup_set_name_var }
 | PASSWORD = { password | @password_variable }
 | { EXPIREDATE = { 'date' | @date_var }
 | RETAINDAYS = { days | @days_var } }
```

备份整个数据库，或者备份一个或多个文件或文件组时使用 BACKUP DATABASE 命令。另外，在完整恢复模式或大容量日志恢复模式下备份事务日志使用 BACKUP LOG 命令。

说明：进行完整数据库备份或差异数据库备份时，SQL Server 会备份足够的事务日志，以便在还原备份时生成一个一致的数据库。

下面对语法中的重要参数进行解释。

- database_name：用于指定备份事务日志、部分数据库或完整的数据库时所用的源数据库。
- <file_or_filegroup>[,...n]：只能与 BACKUP DATABASE 一起使用，用于指定某个数据库文件或文件组包含在文件备份中，或某个只读文件或文件组包含在部分备份中。
- backup_device：指定用于备份操作的逻辑备份设备或物理备份设备。
- DIFFERENTIAL 参数只能与 BACKUP DATABASE 一起使用，指定数据库备份或文件备份应该只包含上次完整备份后更改的数据库或文件部分。差异备份一般会比完整备份占用更少的空间。对于上一次完整备份后执行的所有单个日志备份，

使用该选项可以不必再进行备份。

- COPY_ONLY：指定备份为“仅复制备份”，该备份不影响正常的备份顺序。仅复制备份是独立于定期计划的常规备份而创建的，不会影响数据库的总体备份和还原过程。
- COMPRESSION：表示显式启用备份压缩。NO_COMPRESSION：表示显式禁用备份压缩。
- DESCRIPTION：指定说明备份集的自由格式文本。该字符串最长可以有 255 个字符。
- NAME：指定备份集的名称。名称最长可达 128 个字符。如果未指定 NAME，它将为空。
- PASSWORD：为备份集设置密码。PASSWORD 是一个字符串，但是该功能安全性很低，下一版 SQL Server 可能将删除该功能，所以不建议使用。
- EXPIREDATE：指定备份集到期和允许被覆盖的日期。
- RETAINDAYS：指定必须经过多少天才可以覆盖该备份媒体集。

说明： COMPRESSION|NO_COMPRESSION 是 SQL Server 2008 中新添加的功能选项，默认情况下不进行压缩备份。

例如，当前有数据库 TestDB1，需要将该数据库完整备份到 D:\db1.bak 中，那么对应的 SQL 脚本如代码 6.6 所示。

代码 6.6　完整备份数据库

```
BACKUP DATABASE TestDB1 --备份数据库
TO DISK='D:\db1.bak'
WITH name='TestDB1 Backup',
description='TestDB1 完整备份'
```

备份后的文件就可以转移到其他地方，以备发生异常情况时恢复数据。若要使用前面提到的备份介质，则只需要将备份到硬盘改为备份到介质即可。如将 TestDB1 数据库备份到设备 mydisk1，那么对应的 SQL 脚本如代码 6.7 所示。

代码 6.7　备份数据库到备份设备

```
BACKUP DATABASE TestDB1
TO mydisk1--备份设备
WITH name='TestDB1 Backup',
description='TestDB1 完整备份'
```

除了使用 T-SQL 语句进行备份外，也可以使用 SSMS 进行备份。具体操作如下所述。

（1）在 SSMS 的对象资源管理器中右击要备份的数据库节点（如 TestDB1），在弹出的快捷菜单中选择“任务”选项下的“备份”命令，系统弹出“备份数据库”对话框，如图 6.3 所示。

（2）在数据库下拉列表框中可以选择要备份的数据库，这里默认是 TestDB1，在“备份类型”下拉列表框中可以选择为完整备份、差异备份或日志备份。备份组件中可以选择

是整个数据库备份还是对其中的文件和文件组备份。另外还可以对名称、说明、过期时间等进行设置。

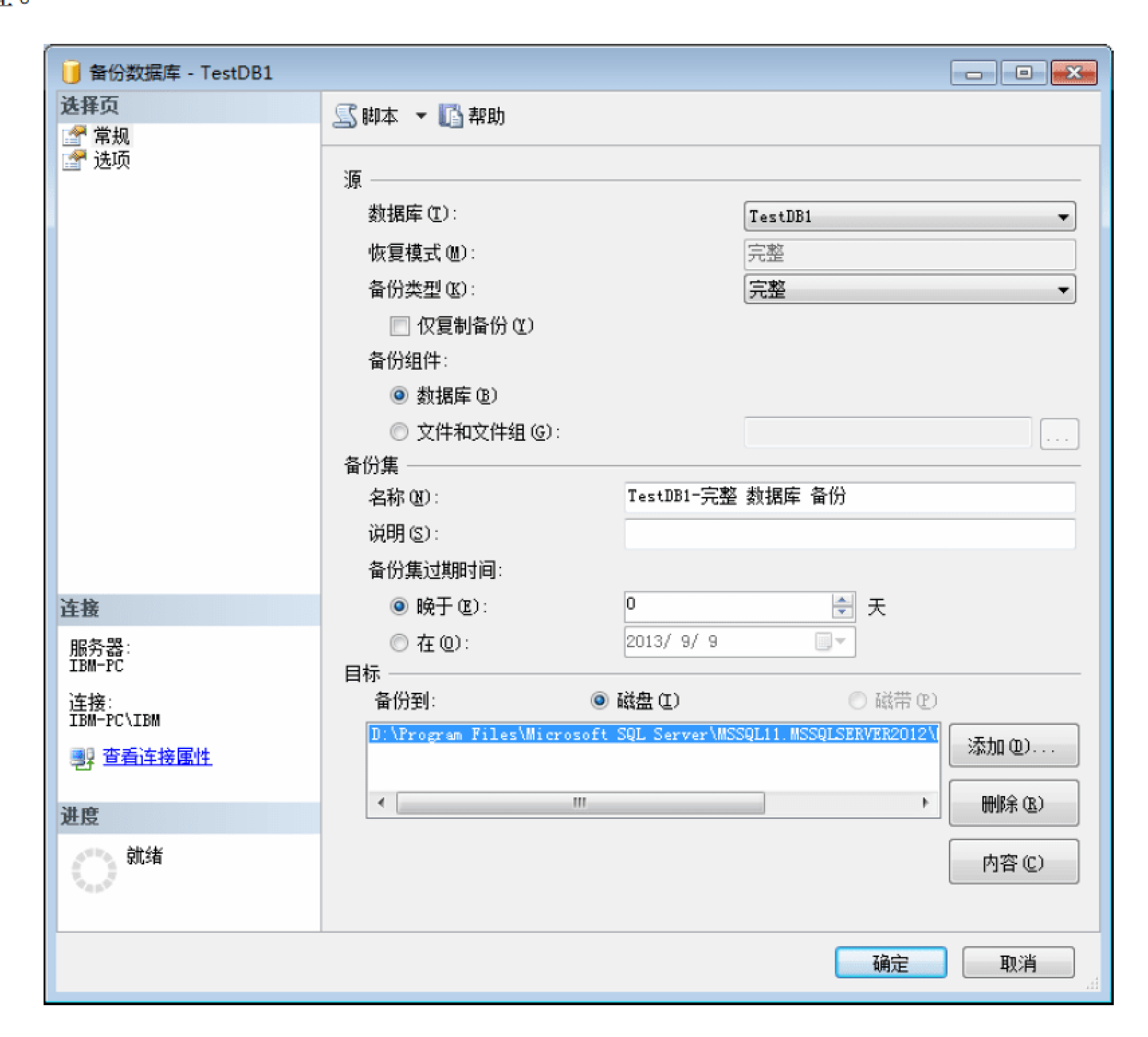


图 6.3　“备份数据库”对话框

（3）单击“删除”按钮，删除默认的备份设备，然后单击“添加”按钮，系统弹出“选择备份目标”对话框，如图 6.4 所示。

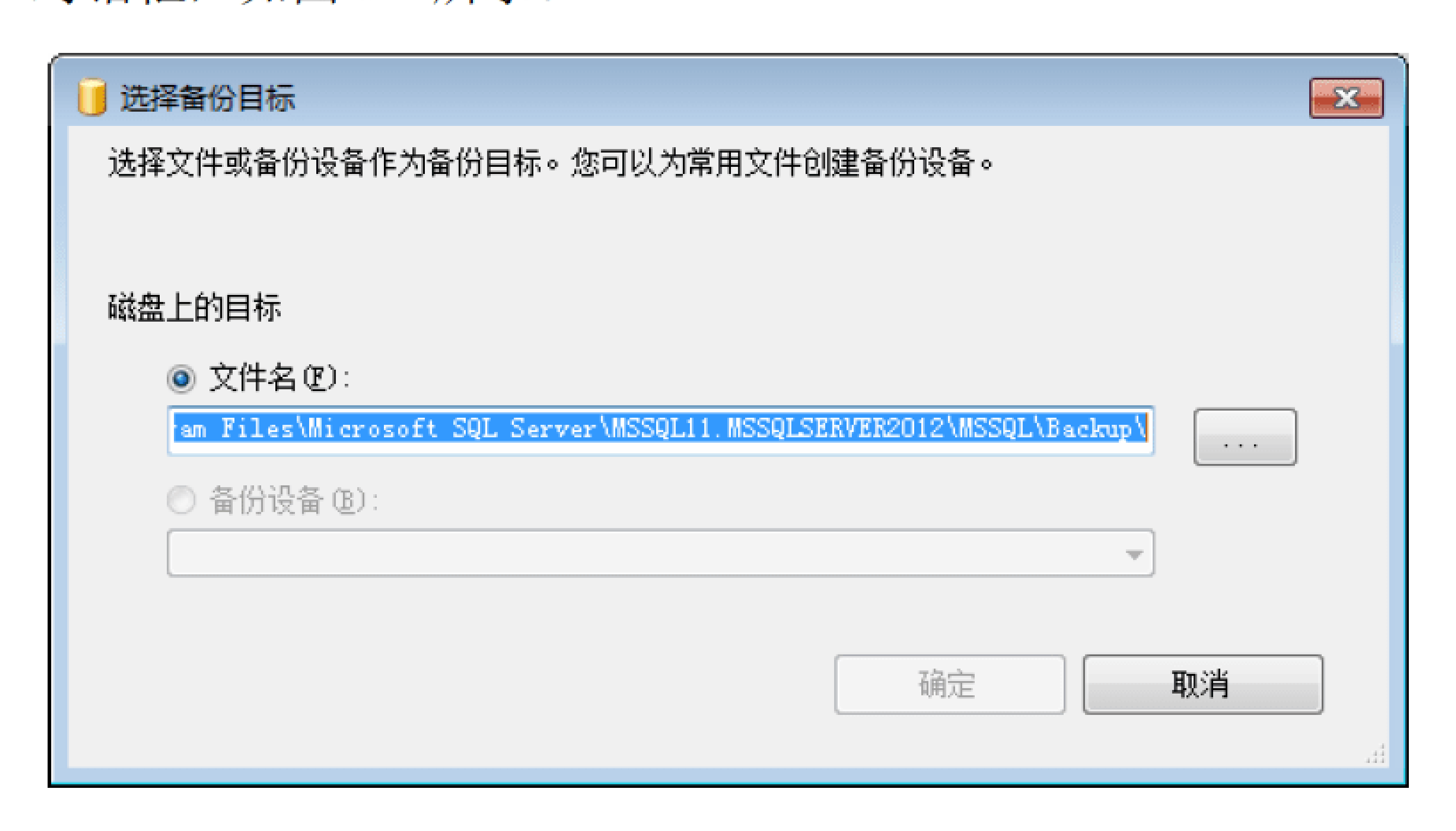


图 6.4　“选择备份目标”对话框

（4）若要直接备份到硬盘上，可以选择“文件名”单选按钮，然后输入要备份的文件路径即可；若是备份到设备可以选择“备份设备”单选按钮，然后从下拉列表框中选择要备份的设备，单击“确定”按钮回到“备份数据库”对话框。

（5）在该对话框中单击“确定”按钮，系统将完成该数据库的备份。

6.2.4　数据库恢复

前面已经做好了数据库的备份，接下来本小节主要是将数据库从备份文件中恢复。与备份数据库类似，恢复数据库使用 RESTORE 命令。其语法格式如代码 6.8 所示。

代码 6.8　RESTORE 语法格式

```
RESTORE DATABASE|LOG { database name | @database name var }
 [ FROM <backup device> [ ,...n ] ]
 [ WITH
   {
       [ RECOVERY | NORECOVERY | STANDBY =
       {standby file name | @standby file name var }
      ]
    | , <general WITH options> [ ,...n ]
      | , <replication_WITH_option>
   | , <change_data_capture_WITH_option>
      | , <service_broker_WITH options>
      | , <point_in_time_WITH_options—RESTORE_DATABASE>
      } [ ,...n ]
 ]
<general_WITH_options> [ ,...n ]::=
--Restore Operation Options
   MOVE 'logical file name in backup' TO 'operating system file name'
         [ ,...n ]
 | REPLACE
 | RESTART
 | RESTRICTED USER
 | FILE = { backup set file number | @backup set file number }
 | PASSWORD = { password | @password_variable }
```

各参数的说明如下所述。

- ❑ RESTORE DATABASE：用于数据库备份的还原，而 RESTORE LOG 用于日志备份的还原。
- ❑ database_name：用于指定要还原到的目标数据库，该数据库名可以是已经存在的数据库名，也可以是不存在的数据库名。对于已存在的数据库名将会被还原所覆盖，而如果指定不存在的数据库名，系统将会创建一个新的数据库来还原。
- ❑ backup_device：指定还原操作要使用的逻辑或物理备份设备。
- ❑ RECOVERY：指示还原操作回滚任何未提交的事务。在恢复进程后即可随时使用数据库。如果既没有指定 NORECOVERY 和 RECOVERY，也没有指定 STANDBY，则默认为 RECOVERY。
- ❑ NORECOVERY：指示还原操作不回滚任何未提交的事务。
- ❑ STANDBY：指定一个允许撤销恢复效果的备用文件。STANDBY 选项可以用于脱机还原（包括部分还原），但不能用于联机还原。
- ❑ FILE：由于可以在相同的备份介质上备份多次，该选项可选择恢复特定的版本。如果不提供该值，SQL Server 将默认从最新的备份中恢复。
- ❑ MOVE：允许从最初备份数据库的地方恢复到不同物理文件中。

例如，现在有数据库备份文件 C:\db1.bak，需要将该数据库备份还原为数据库 db1，该

数据库文件在 C:\Data 目录下，那么对应的 SQL 脚本如代码 6.9 所示。

代码 6.9　还原数据库

```
RESTORE DATABASE [db1]
FROM  DISK = 'C:\db1.bak'
WITH  FILE = 1,
MOVE 'TestDB1' TO 'C:\DATA\db1.mdf',        --还原后的数据文件路径
MOVE 'TestDB1_log' TO 'C:\DATA\db1_1.ldf'  --还原后的日志文件路径
```

在 SSMS 中恢复数据库的操作主要有以下几步。

（1）在 SSMS 的对象资源管理器中右击“数据库”节点，在弹出的快捷菜单中选择“还原数据库”选项，系统将弹出“还原数据库”对话框，如图 6.5 所示。

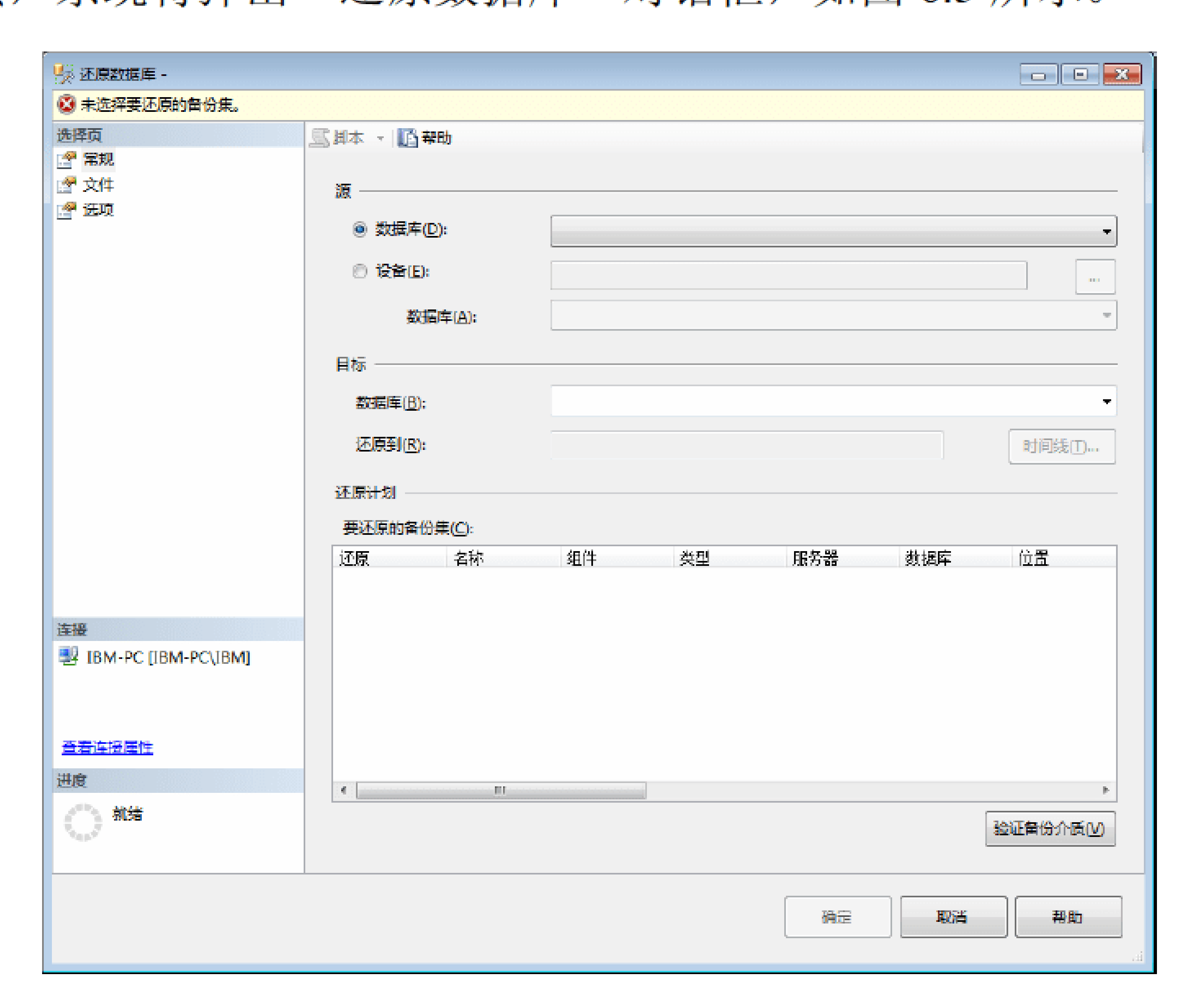


图 6.5　“还原数据库”对话框

（2）在“目标”选项下的“数据库”下拉列表框中输入要还原的数据库的名称，如 db2。

（3）选择“源设备”单选按钮，然后单击文本框旁边的下三角按钮，系统将弹出“选择备份设备”对话框，如图 6.6 所示。

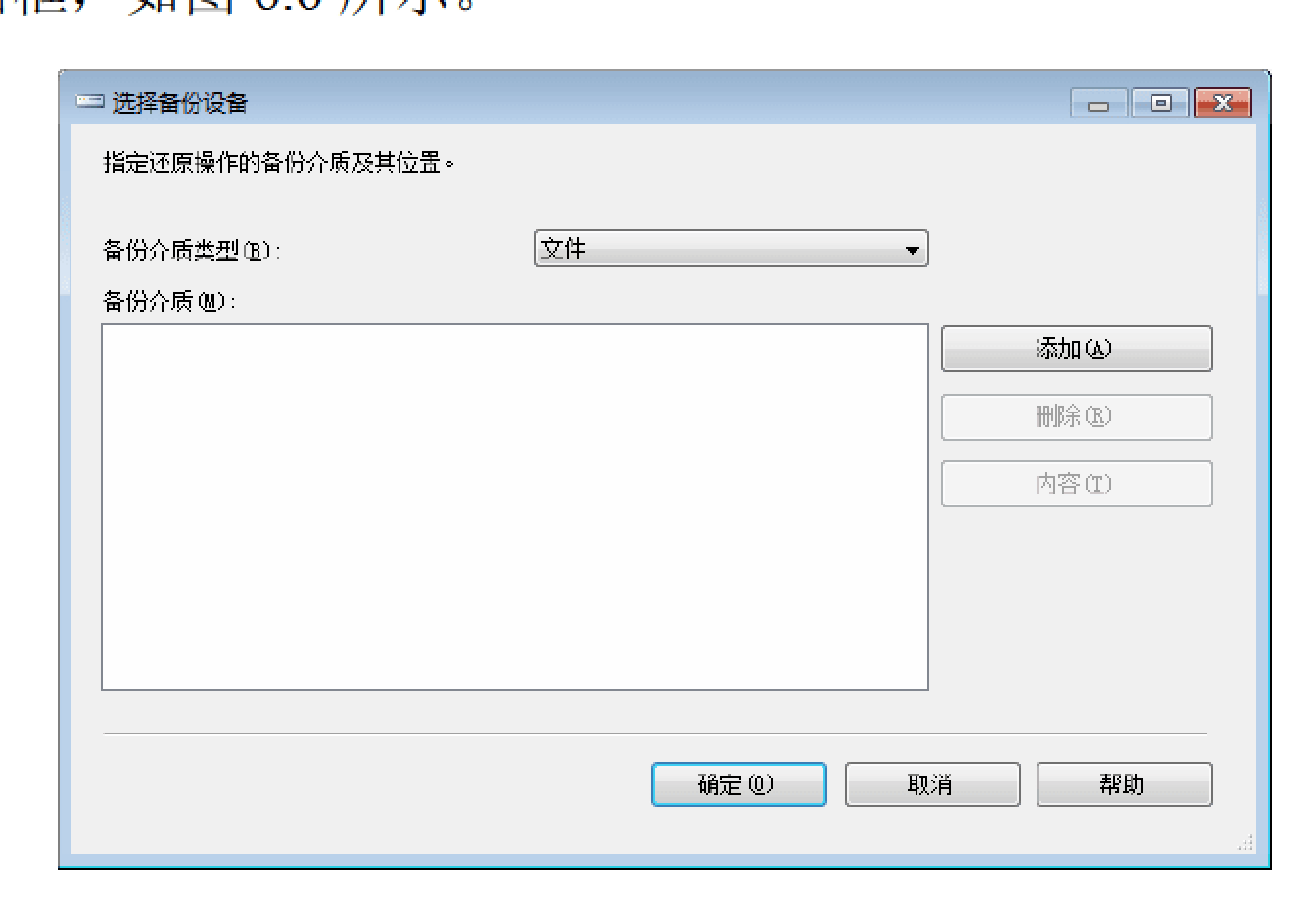


图 6.6　“选择备份设备”对话框

（4）在其中单击“添加”按钮添加要还原的数据库备份的路径，然后单击“确定”按钮，系统回到“还原数据库”对话框。

（5）选中要还原的备份集，然后单击“选项”选项，系统切换到“选项”选项卡，如图 6.7 所示。

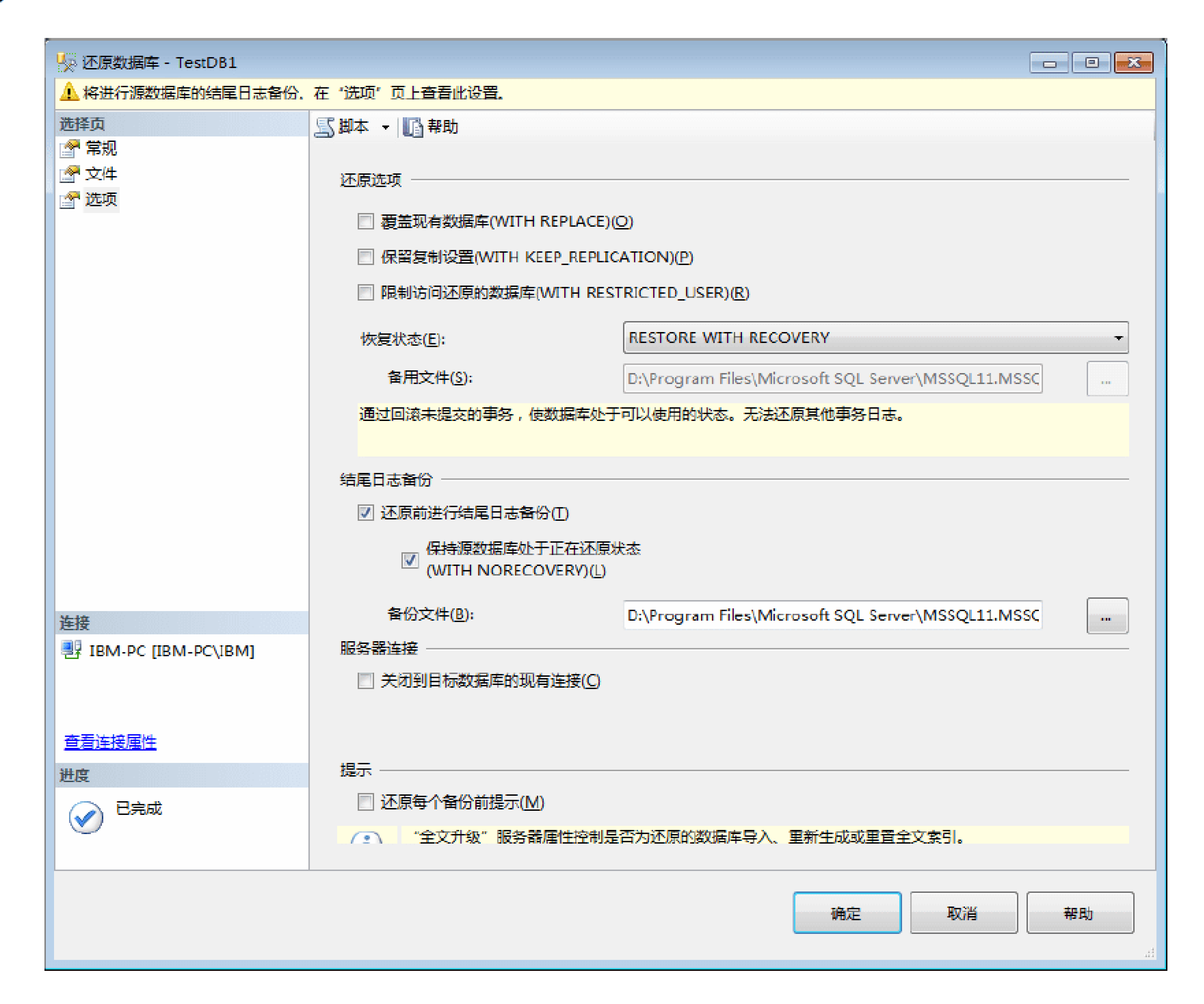


图 6.7　“选项”选项卡

（6）如果是还原数据库覆盖现有数据库，那么将“覆盖现有数据库”复选框选中。

（7）如果需要修改还原后数据库文件的数据文件和日志文件的存放路径，可以修改中间“还原为”列中的配置。

（8）如果要修改恢复状态，可以选中对应的单选按钮。

（9）配置完成后单击“确定”按钮，系统即将备份还原到数据库中。

6.2.5　恢复模式

备份和还原操作是在“恢复模式”下进行的。恢复模式是数据库中的一个属性，用于控制数据库备份和还原的基本行为。恢复模式不仅简化了恢复计划，而且还简化了备份和还原的过程，同时明确了系统要求之间的平衡，也明确了可用性和恢复要求之间的平衡。SQL Server 2012 中提供了完整、简单和大容量日志等 3 种恢复模式。

1．完整恢复模式

完整恢复模式是 SQL Server 默认的恢复模式，在出现数据库文件损坏时丢失数据的风险最小。如果一个数据库被设置为完整恢复模式，所有的操作都会在日志中完整地记录下来。完整恢复模式下不仅会记录 INSERT 执行的插入操作、UPDATE 执行的更新操作，以

及 DELETE 执行的删除操作，同时也会记录 bcp 或 BULK INSERT 等批量操作插入的每一行到其事务日志中。

另外，在完整恢复模式下，SQL Server 也会完整记录 CREATE INDEX 操作。当从包含创建索引的事务日志中恢复时，不需要再重建索引，恢复操作也会进行的十分迅速。由于有如此多的信息需要记录在日志中，所以在完整恢复模式下，日志的膨胀速度很快，所以也需要执行定期的日志备份和清理。

如果遇到数据库文件被损坏的情况，而该数据库是处于完整恢复模式下，而且在进行了完整数据库备份后一直在做定期的事务日志备份，那么可以将数据库恢复到最后一个日志备份的时间点状态。如果是数据库的数据文件被破坏，而日志文件可用，那么可以将该数据库恢复到数据文件被破坏前的最后一个日志记录时间点的状态。

完整恢复模式能够尽量减少数据恢复时的数据丢失情况，但是其缺点就是会生成很大的事务日志，所以需要更多的磁盘空间和更多的日志维护时间。

2. 大容量日志恢复模式

大容量日志恢复模式简单地记录了大多数大容量操作（如 BULK INSERT、CREATE INDEX、SELECT INTO 等），而不是每一条数据操作都完整记录，但是完整地记录了其他事务。当在该恢复模式下的数据库中执行了大容量操作时，SQL Server 只会记录该操作曾经发生过和关于该操作分配空间的信息。

因为大容量操作只进行了简单的记录，所以这些大容量操作比在完整恢复模式下执行要快很多。如果数据库是在大容量日志恢复模式下，而实际上并没有执行过大容量操作，因为该日志将会记录数据库修改的完整信息，所以可以将数据库还原到任一时间。但是，如果执行了大容量操作，大容量日志恢复模式增加了这些操作丢失数据的风险。

3. 简单恢复模式

简单恢复模式提供了最简单的备份恢复策略。简单恢复模式简略地记录了大多数事务，所记录的信息只是为了确保在系统崩溃或还原数据备份之后的一致性。由于旧的事务已经提交，已不再需要其日志，所以日志将会被截断。由于日志经常会被截断，所以数据库日志文件并不会像其他两种模式那样一直膨胀，而是一直保存在大约 10MB 的大小。

在简单恢复模式下所能进行的备份类型就是那些不需要日志备份的类型，这些类型的备份有：完整数据库备份、增量备份、部分完整备份、部分增量备份和针对只读文件组的文件组备份。

简单恢复模式下并不是不记录日志，所谓“简单”是指备份策略中不需要担心日志备份。在简单恢复模式下，事务日志将会被截断，也就是说，不活动的日志将会被删除。因为经常会发生日志截断，所以不能进行日志备份，如果试图进行日志备份，系统将抛出异常。

简单恢复模式由于经常会发生日志截断，并没有完整记录和保存事务日志，所以在数据库恢复时只能恢复到上一次数据库备份的数据，而备份以后的数据将无法进行恢复，造成最近数据的丢失。

注意：简单恢复模式并不适合于生产系统，因为对生产系统而言，丢失最新的更是无法接受的。微软建议使用完整恢复模式。

6.3　数据文件的转移

当进行系统维护之前、发生硬件故障之后或者更换了系统硬件时就需要将数据库进行转移，这时就需要用到数据库的分离和附加。复制数据库是创建一个备份开发环境或测试环境常用的方法，复制数据库也可以通过分离和附加来完成。

6.3.1　分离数据库

分离数据库是指将数据库从 SQL Server 实例中删除，但数据库在其数据文件和事务日志文件中保持不变。之后，就可以使用这些文件将数据库附加到任何 SQL Server 实例，包括分离该数据库的服务器。

分离数据库之前必须要保证没有用户正在使用该数据库。如果发现无法终止已存在的连接，可以使用 ALTER DATABASE 命令将数据库设置为单用户模式。分离数据库中没有不完整的事务，也没有留在内存中的脏数据页面。如果不满足这些条件，分离操作就不会成功。

注意：在未分离数据库也未关闭数据库服务的情况下是无法复制数据库文件的，尝试复制文件时系统将会抛出异常：文件正在被使用。所以必须关闭数据库服务或者分离数据库后才能进行数据库文件的复制。

一旦数据库被分离，从 SQL Server 角度看与删除该数据库并没有什么不同。在 SQL Server 中不会留下该数据库的痕迹。

说明：删除数据库和分离数据库都会从 SQL Server 中清除该数据库的所有痕迹，但是删除数据库会从操作系统中删除数据库对应的物理文件，而分离数据库后数据库的文件仍然存在。

在 T-SQL 中分离数据库使用系统存储过程 sp_detach_db。该存储过程的语法，如代码 6.10 所示。

代码 6.10　sp_detach_db 语法

```
sp_detach_db [ @dbname= ] 'database_name'
    [ , [ @skipchecks= ] 'skipchecks' ]
    [ , [ @keepfulltextindexfile = ] 'KeepFulltextIndexFile' ]
```

其中，database_name 为要分离的数据库的名称。skipchecks 指定跳过还是运行 UPDATE STATISTICS。若要跳过 UPDATE STATISTICS，则为 true。若要显式运行 UPDATE STATISTICS，则指定 false。KeepFulltextIndexFile 指定在数据库分离操作过程中不会删除与所分离的数据库关联的全文索引文件，默认为 true。

例如有数据库 TestDB1，现在需要将该数据库分离，而该数据库可能有用户正在连接，需要先将该数据库设置为单用户模式，然后再分离数据库。其操作执行的 SQL 脚本如代码 6.11 所示。

代码 6.11　分离数据库

```
USE master;
GO
ALTER DATABASE TestDb1 --修改数据库为单用户模式
SET SINGLE USER;
GO
EXEC sp_detach_db 'TestDb1','true' --分离数据库
```

在 SSMS 中分离数据库的操作主要有以下几步。

（1）在 SSMS 的对象资源管理器中右击需要分离的数据库，在弹出的快捷菜单中选择“任务”选项下的“分离”命令，系统将弹出“分离数据库”对话框，如图 6.8 所示。

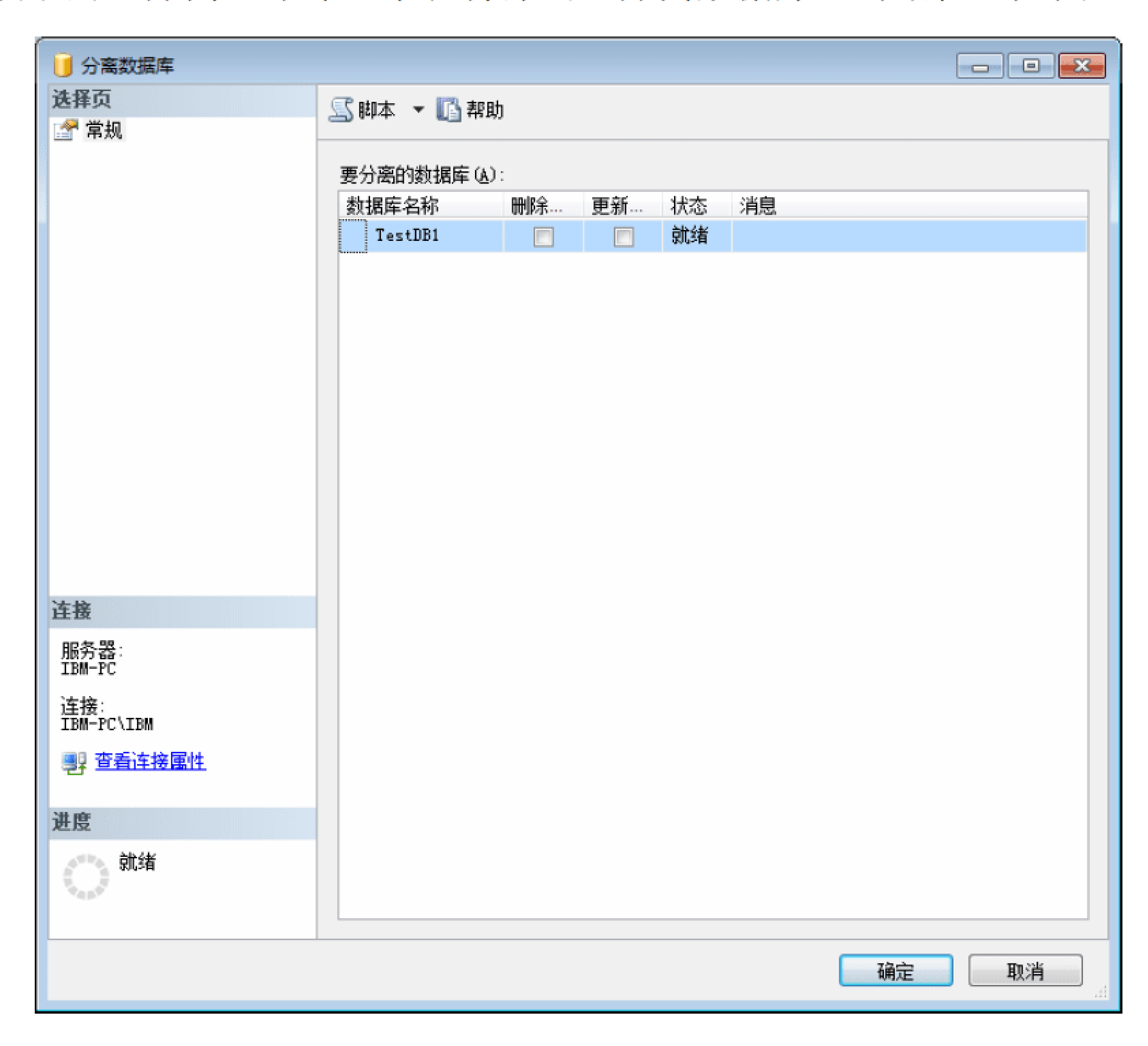


图 6.8　“分离数据库”对话框

（2）从该对话框中可以看到当前还有一个活动连接，所以必须选中“删除连接”复选框，若需要更新统计信息，可以选中“更新统计信息”复选框。

（3）单击“确定”按钮，系统即可完成数据库的分离操作。

6.3.2　附加数据库

在分离数据库后，就可以将数据库文件转移或复制到其他地方，然后通过附加数据库的方式来还原该数据库。

为了附加一个数据库，可以使用系统存储过程 sp_attach_db，但是现在已经不再推荐使用，而且可能会在将来的版本中删除该存储过程，微软建议使用带 FOR ATTACH 选项的 CREATE DATABASE 命令。

系统存储过程 sp_attach_db 限制了最多只能附加 16 个文件，而 CREATE DATABASE 则没有这个限制，事实上可以为每个数据库指定多达 32 767 个文件和 32 767 个文件组。使用 CREATE DATABASE 命令附加数据库的语法如代码 6.12 所示。

代码 6.12　CREATE DATABASE 附加数据库语法

```
CREATE DATABASE database name
ON <filespec> [ ,...n]
FOR {ATTACH | ATTACH_REBUILD_LOG}
```

其中，database_name 就是附加数据库后的数据库名，该名可以与要附加的数据库文件的原数据库名不同。filespec 就是要附加的数据库文件，而 ATTACH 表示附加数据库文件，而 ATTACH_REBUILD_LOG 则表示在附加数据库时重建数据库的日志。例如前面已经将数据库 TestDB1 分离了，现在需要将该数据库附加回去并命名为数据库 TestDB2，那么对应的 SQL 脚本如代码 6.13 所示。

代码 6.13　附加数据库

```
USE master;
GO
CREATE DATABASE TestDB2
ON （FILENAME = 'D:\DATA\TestDB1.mdf'）
FOR ATTACH --附加操作
```

注意： SQL Server 2012 可以附加 SQL Server 2000 及以后的数据库文件，但是对于 SQL Server 2000 以前的数据库文件则不能附加。

如果一个可读写的数据库含有当前不可用的日志文件，如果该数据库在附加操作之前，在没有用户和活动事务的情况下被关闭，那么 FOR ATTACH 会重建该日志文件并更新主文件中有关日志的信息。如果该数据库是只读数据库，那么就不能更新主文件中关于日志文件的信息，所以也就不能重建日志文件，也不能附加成功。同样，如果是只读数据库，那么就不能在丢失日志文件的情况下使用 ATTACH_REBUILD_LOG 选项来重建日志文件。

技巧： 如果是需要将生存环境中的数据库复制到测试环境中，那么只需要复制数据文件，而不需要复制庞大的日志文件。然后在测试环境中使用 ATTACH_REBUILD_LOG 重建日志。

使用 SSMS 附加数据库的操作主要有以下几步。

（1）在 SSMS 的对象资源管理器中右击“数据库”节点，在弹出的快捷菜单中选择“附加”选项，系统将弹出“附加数据库”对话框，如图 6.9 所示。

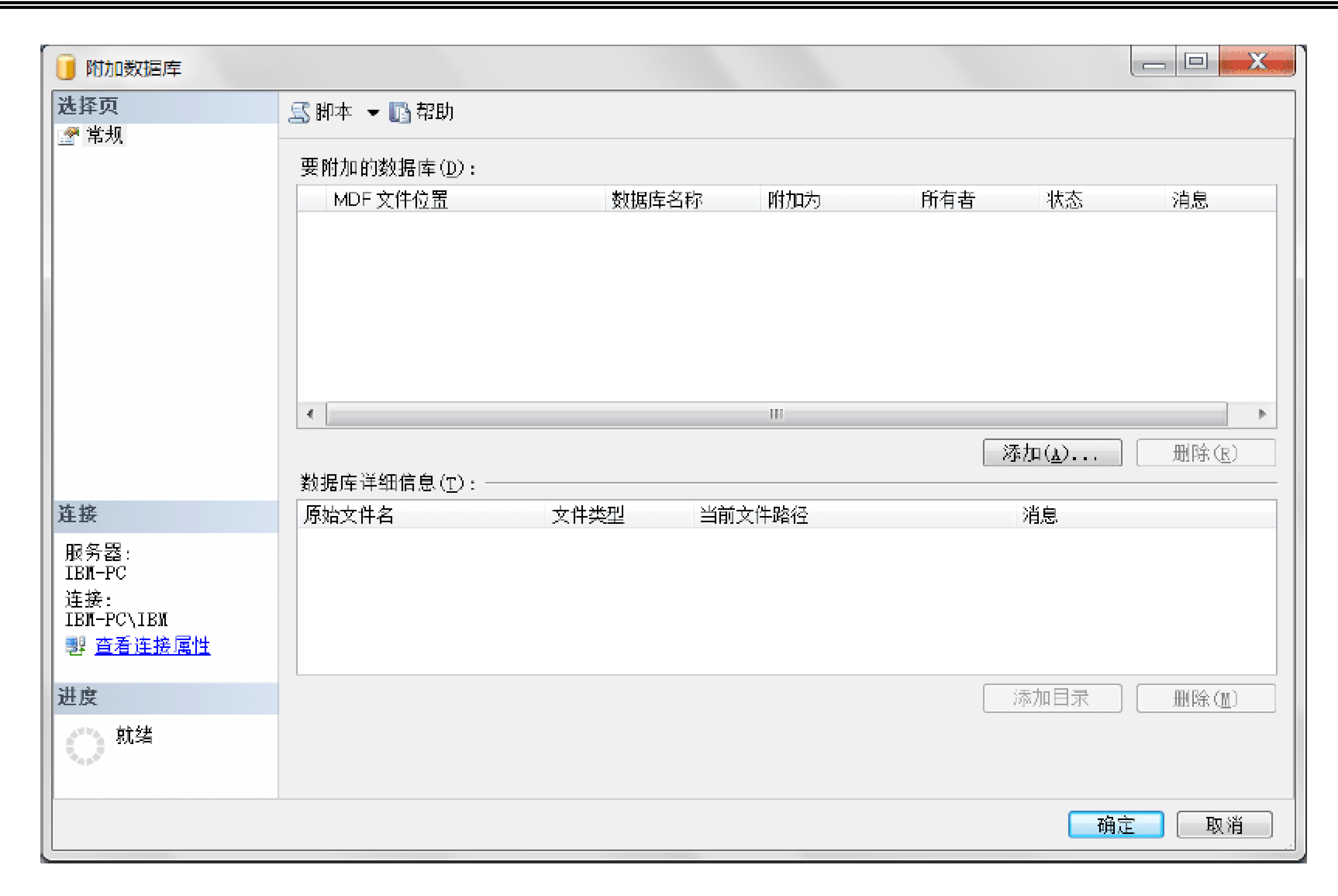


图 6.9 “附加数据库”对话框

（2）单击“添加”按钮，添加要进行附加的数据库主文件（mdf 文件），系统将根据数据库主文件自动找到对应的日志文件。

（3）如果找到的文件路径有误或者没有日志文件，那么单击“数据库详细信息”下面的“删除”按钮，删除有误的文件。

（4）单击“确定”按钮，完成数据库的附加操作。

注意：SQL Server 允许低版本的数据库在 SQL Server 2012 中还原或附加，但是一旦还原或附加后即使不做任何修改，再重新将这个数据库备份或分离将无法再在低版本的 SQL Server 中还原或附加。

6.4　数据库快照

数据库快照是数据库的只读、静态视图，是 SQL Server 2005 中添加的新功能。数据库快照提供了快速、简洁的一种数据库另类备份操作。多个快照可以位于同一个源数据库中，并且可以作为数据库始终驻留在同一服务器实例上。创建快照时，每个数据库快照在事务上与源数据库一致。在被数据库所有者显式删除之前，快照始终存在。

快照可用于报表。另外，如果源数据库出现用户错误，还可将源数据库恢复到创建快照时的状态。丢失的数据仅限于创建快照后数据库更新的数据。

6.4.1　数据库快照原理

数据库快照在数据页级运行。如果对数据库建立了快照，在第一次修改源数据库页之前，系统先将原始页从源数据库复制到快照。此过程称为“写入时复制操作”。快照将存储原始页，保留它们在创建快照时的数据记录。对已修改页中的记录进行后续更新不会影

响快照的内容。对要进行第一次修改的每一页重复此过程，这样，快照将保留自创建快照后经修改的所有数据记录的原始页。

SQL Server 中使用了一种叫做“稀疏文件”的文件来存储复制的原始页。最初，稀疏文件实质上是空文件，不包含用户数据并且未被分配存储用户数据的磁盘空间。对于每一个快照文件，SQL Server 创建了一个保存在高速缓存中的比特图，数据库文件的每一个页面对于一个比特位，表示该页面是否以及被复制到快照中。当源数据库发生改变时，SQL Server 会查看比特图来检查该页面是否已经被复制，如果没有被复制，那么马上将其复制到快照中，然后再更新源数据库，这种操作叫写入时复制（copy-on-write）操作。当然，如果该页已经被复制到快照文件中了就不需要再重复复制了。

注意： 快照只能在 NTFS 格式的盘上创建，因为该格式是唯一支持稀疏文件技术的文件格式。

随着源数据库中更新的页越来越多，快照文件中保存的页也越来越多，快照文件的大小也不断增长。创建快照时，稀疏文件占用的磁盘空间很少。然而，由于数据库随着时间的推移不断更新，稀疏文件会增长为一个很大的文件。如图 6.10 所示，修改源数据库数据时，系统将复制源数据库中修改对应的数据页到数据库快照中。

对于用户而言，数据库快照似乎始终保持不变，因为对数据库快照的读操作始终访问原始数据页，而与页驻留的位置无关。当一个查询从快照中读取数据，它首先通过比特图来判断需要的页面是否已经存在于快照文件中，或者仍然在源数据库中。如图 6.11 显示了一个快照查询对数据库的访问情况。源数据库的 9 个页面被访问到，有一个页面是通过快照来访问的，因为该页面已经被更新过了。

无论是从稀疏文件中读取还是从源数据库中读取，无论处于何种隔离级别之下，都不需要使用任何锁，这是数据库快照的一大优点。

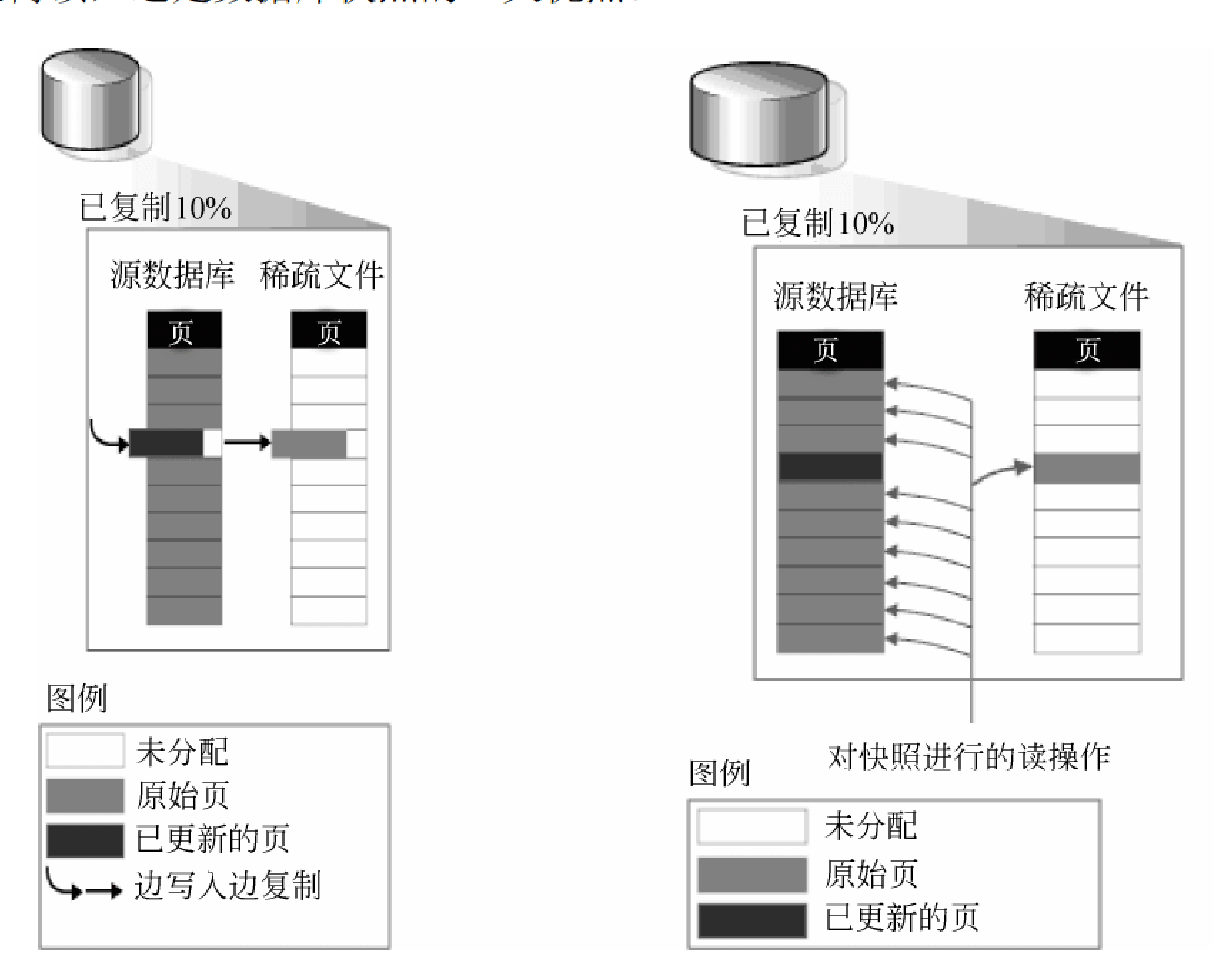


图 6.10　包含来自源数据库中一个页的数据库快照　　图 6.11　读取数据库快照

前面提到，比特图是保存在高速缓存中，而不是在数据库文件中，所以它总是可以随时使用。当 SQL Server 关闭时，比特图会丢失，然后再在数据库启动时进行重建。当 SQL Server 被访问时它会判断每一个页面是否在稀疏文件中，然后将这些信息保存在比特图中供将来使用。

6.4.2　建立数据库快照

任何具有创建数据库权限的用户都可以创建数据库快照。数据库快照功能只有 SQL Server 企业版才可用。

创建数据库快照之前，考虑如何命名是非常重要的。每个数据库快照都需要一个唯一的数据库名称，而且数据库快照的名称不能和其他数据库名重复。为了便于管理，一般情况下在数据库快照命名中可以包含源数据库的名称、创建快照的日期时间、序号或其他一些信息以区分给定数据库上的多个快照。

SQL Server 中只能使用 T-SQL 语句来创建快照，而不支持使用 SSMS 进行可视化的快照创建操作。窗口数据库快照使用带有 AS SNAPSHOT 的 CREATE DATABASE 命令。创建数据库快照的语法如代码 6.14 所示。

代码 6.14　创建数据库快照语法

```
CREATE DATABASE db_snapshot_name ON
(Name='name',FileName='file_path')
AS SNAPSHOT OF db_name
```

其中的 db_snapshot_name 是要创建的快照的名字，name 用于指定要备份的源数据库的数据文件的逻辑名称，file_path 用于指定快照稀疏文件的物理路径，db_name 便是要用于创建快照的源数据库。稀疏文件以 64KB 为单位增长，因此磁盘上稀疏文件的大小总是 64KB 的倍数。例如现在需要对 AdventureWorks2012 数据库建立快照，则对应的 T-SQL 语句如代码 6.15 所示。

代码 6.15　为 AdventureWorks 建立数据库快照

```
USE master
GO
CREATE DATABASE AdventureWorks_Snapshot --数据库快照名
ON(Name='AdventureWorks2012_Data',FileName='D:\SQL
Data\AdventureWorks2012.ss')
--快照文件路径
AS snapshot OF AdventureWorks2012
```

建立数据库快照后，对应的路径上会建立与源数据库数据文件大小相同的稀疏文件，该文件的大小虽然与源数据库数据文件大小相同，但是由于源数据库还未做数据更改，也没有数据复制到快照文件中，所以其占用空间却十分小。如图 6.12 所示为 AdventureWorks2012 数据库的快照文件的属性，其大小为 205MB，但是占用空间却只有 3.12MB。

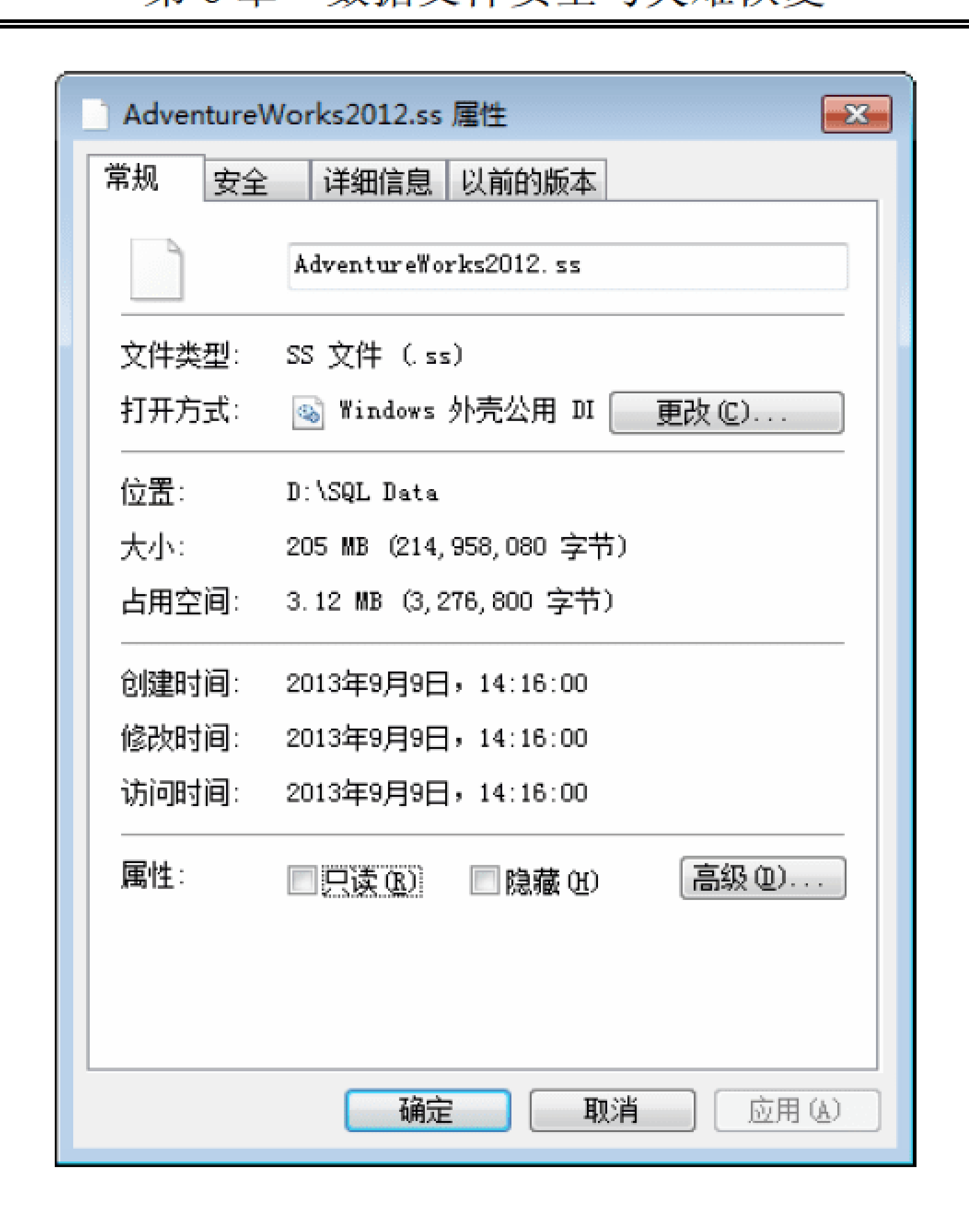


图 6.12　快照文件的属性

6.4.3　管理数据库快照

如果为一个数据库建立了快照，那么该数据库将无法删除、分离或还原。如果将一个数据库切换到离线状态，那么其快照也会被自动删除。数据库快照的一个基本作用就是用于数据库的备份，当需要将源数据库还原到快照时的状态时，可以使用 RESTORE 命令，从数据库快照中还原数据库的语法格式如代码 6.16 所示。

代码 6.16　RESTORE 从数据库快照中还原数据库的语法

```
RESTORE DATABASE <database_name>
FROM DATABASE_SNAPSHOT  = <database_snapshot_name>
```

例如先对 AdventureWorks2012 数据库进行修改，然后再把该数据库从前面建立的快照 AdventureWorks_Snapshot 中恢复过来，那么对应的 SQL 语句如代码 6.17 所示。

代码 6.17　从快照中恢复数据库

```
USE AdventureWorks2012;
GO
DELETE FROM dbo.DatabaseLog --删除数据
GO
SELECT COUNT (*)  --这里返回 0 行数据，因为都已经被删除了
FROM dbo.DatabaseLog
GO
USE master
GO
RESTORE DATABASE AdventureWorks2012  --从快照中还原数据库
FROM DATABASE_SNAPSHOT  = 'AdventureWorks_Snapshot'
GO
USE AdventureWorks2012;
```

```
GO
SELECT COUNT (*)   --这里返回了数据而不是 0，因为数据库已经被还原回来了
FROM dbo.DatabaseLog
```

注意： 当同一个数据库存在多个数据库快照时，是不能还原其中的任何一个快照的，所以必须把除了要恢复的快照保留外其他快照全部删除，然后才能从快照中恢复数据库。删除数据库快照使用 DROP DATABASE 命令。

需要注意下面这些与数据库快照有关的附加注意事项。

- 不能对 model、master 和 tempdb 数据库创建快照。
- 一个快照会从其源数据库中继承安全约束，而且由于快照是只读的，所以不能改变快照中的权限。
- 如果从源数据库中删除用户，该用户会继续保留在快照中。
- 不能备份和还原快照，但能正常备份源数据库。
- 不能分离和附加快照。
- 数据库快照不支持全文索引，全文目录不会从源数据库传播到快照中。

6.5　数据库镜像

数据库镜像是用于提高数据库可用性的主要软件解决方案。数据库镜像大大提高了可用性，并为故障转移群集或日志传送提供了一种易于管理的替代方案或补充方案。本节将主要对数据库镜像的原理和实现进行讲解。

6.5.1　数据库镜像概论

数据库镜像是在 SQL Server 2005 中添加的一个新功能，镜像基于每个数据库实现，并且只适用于使用完整恢复模式的数据库。不支持对简单恢复模式和大容量日志恢复模式数据库进行数据库镜像。数据库镜像可使用任意支持的数据库兼容级别。

注意： 不能镜像 master、msdb、tempdb 或 model 等系统数据库。

数据库镜像实际上就是在不同的 SQL Server 数据库引擎服务器实例上维护一个数据库的两个副本。通常在正式企业环境中，这些服务器实例驻留在不同的计算机上，甚至可能在不同地域的计算机上。其中一个服务器实例是直接被客户端连接和使用的，被称为“主服务器”，另一个服务器实例则根据镜像会话的配置和状态，充当热备用或温备用服务器，并不被客户端连接，称为“镜像服务器”。同步数据库镜像会话时，数据库镜像提供热备用服务器，可支持在已提交事务不丢失数据的情况下进行快速故障转移。未同步会话时，镜像服务器通常用做备用服务器，这种情况下可能造成数据丢失。

在“数据库镜像会话”中，主体服务器和镜像服务器扮演互补的伙伴角色：“主体角色”和“镜像角色”。在任何给定的时间，都是一个伙伴扮演主体角色，另一个伙伴扮演镜像角色。每个服务器之间的角色是可以互换的。拥有主体角色的伙伴称为“主体服务器”，

其数据库副本为当前的主体数据库。拥有镜像角色的伙伴称为“镜像服务器”，其数据库副本为当前的镜像数据库。当在生产环境中部署数据库镜像时，则主体数据库即为“生产数据库”。

数据库镜像的原理就是使用日志的方式尽快将对主体数据库执行的每项插入、更新和删除操作“重做”到镜像数据库中。重做通过将每个活动事务日志记录发送到镜像服务器来完成，每个提交的事务在主服务器中一方面是将事务写入日志当中，另一方面也将事务提交到镜像服务器中，由镜像服务器又将事务记录到日志当中。与逻辑级别执行的复制不同，数据库镜像在物理日志记录级别执行。在数据库镜像中提交一个修改命令的过程如图 6.13 所示。

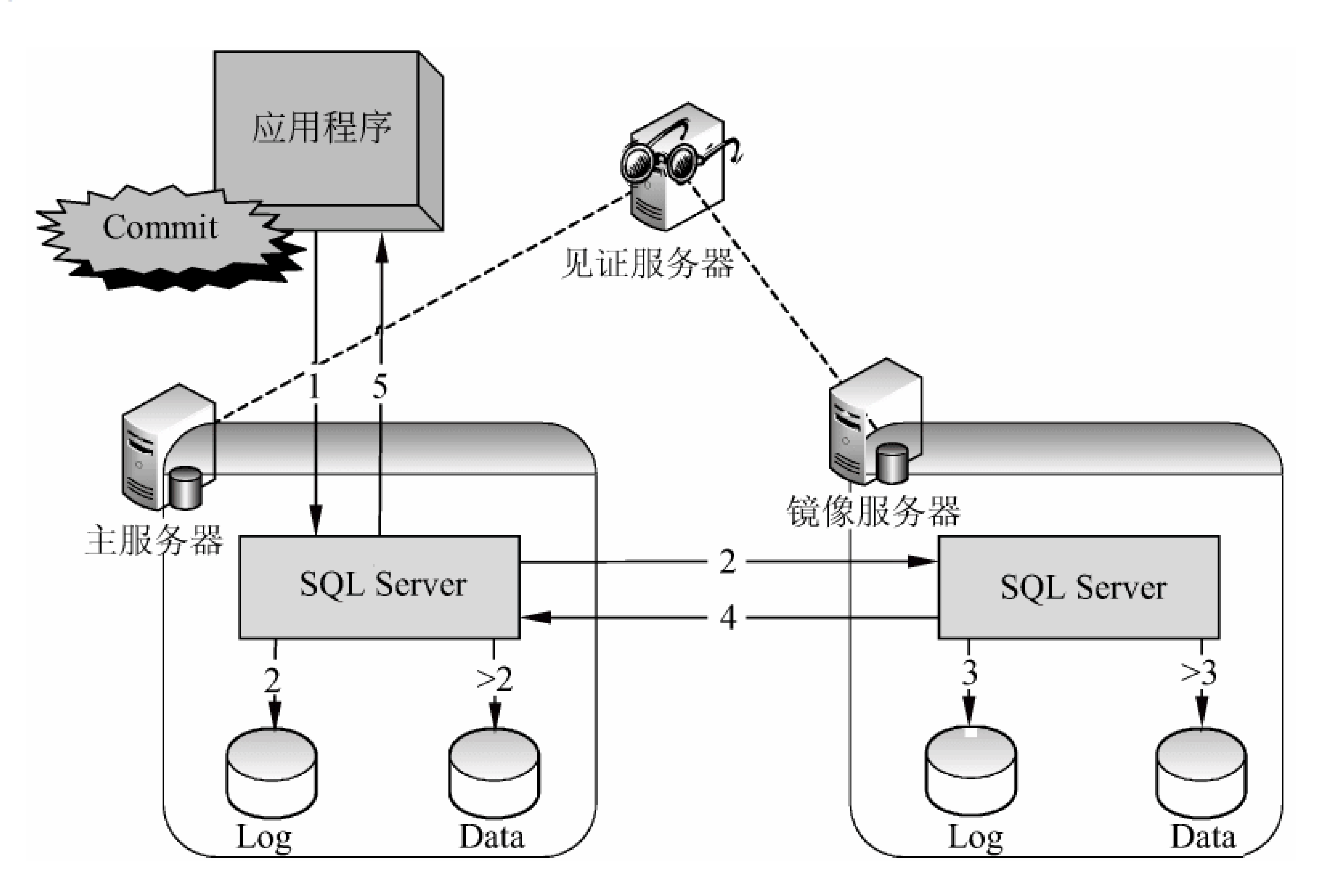


图 6.13　数据库镜像原理

在数据库镜像中，通常可以使用一个称为“角色切换”的过程来互换主体角色和镜像角色。当发生自动故障转移或者人为进行角色切换时，系统将主体角色转换给镜像服务器，原主服务器离线或者转换为镜像服务器。在角色切换中，镜像服务器充当主体服务器的“故障转移伙伴”。

进行角色切换时，镜像服务器将接管主体角色，并使其数据库的副本在线以作为新的主体数据库。原主体服务器如果还存在的话将充当镜像角色，并且其数据库将变为新的镜像数据库。这些角色可以反复地来回切换。SQL Server 中存在以下 3 种角色切换形式。

- 自动故障转移：这要求使用高安全性模式并具有镜像服务器和见证服务器。数据库必须已同步，并且见证服务器必须连接到镜像服务器。见证服务器通过心跳线与主服务器和镜像服务器进行连接，验证给定的伙伴服务器是否已启动并运行。如果镜像服务器与主体服务器断开连接，但见证服务器仍与主体服务器保持连接，则镜像服务器无法启动故障转移。
- 手动故障转移：这要求使用高安全性模式。 伙伴双方必须互相连接，并且数据库必须已同步。
- 强制服务（可能造成数据丢失）：在高性能模式和不带自动故障转移功能的高安全性模式下，如果主服务器出现故障而镜像服务器可用，则可以强制服务运行。

使用数据库镜像有如下优点：

- 增强数据保护功能。数据库镜像运行模式是高安全性或高性能时提供完整或接近完整的数据冗余。数据库镜像伙伴会在无法读取页时向其他伙伴请求新副本，如果此请求成功，则将以新副本替换不可读的页，这便实现了自动尝试解决某些阻止读取数据页的错误。
- 提高数据库的可用性。在具有自动故障转移功能的高安全性模式下，如果主服务器发生故障，自动故障转移可快速使镜像服务器切换到主服务器角色，将其中的数据库的备用副本联机而不会丢失数据。如果在其他运行模式下，数据库管理员可以选择强制服务（可能丢失数据），以替代数据库的备用副本。
- 提高生产数据库在升级期间的可用性。如果对数据库服务器进行升级时需要重启服务器或停止数据库服务，则可以先对服务器进行升级，升级完成后再手动将镜像服务器切换为主服务器，然后对切换下来的服务器进行升级，这种滚动升级方式只导致了一个故障转移停机时间，而不会造成长时间数据库停止服务。

6.5.2　数据库镜像模式

数据库镜像会话以同步操作或异步操作运行。在异步操作下，事务传送到镜像服务器后不需要等待镜像服务器返回，将日志写入磁盘的消息便可提交，这样可最大程度地提高性能。在同步操作下，事务将在伙伴双方处提交，在镜像服务器返回消息确认日志已经成功写入后，主服务器才返回客户端消息，这样会延长事务滞后时间。SQL Server 2012 支持以下两种镜像运行模式。

- 高安全性模式，它支持同步操作。在高安全性模式下，当会话开始时，镜像服务器将进行初始化，将使镜像数据库尽快与主体数据库同步。一旦同步了数据库，事务将按照同步的方式在伙伴双方处提交，这会延长事务滞后时间。
- 高性能模式，是异步运行。镜像服务器尝试与主体服务器发送的日志记录保持同步，但主服务器并不会等待镜像服务器返回成功消息。镜像数据库可能稍微滞后于主体数据库。但是，数据库之间的时间间隔通常很小。如果主体服务器的工作负荷过高或镜像服务器系统的负荷过高，则时间间隔会增大。

在异步运行高性能模式中，主体服务器向镜像服务器发送日志记录之后，会立即再向客户端发送一条确认消息。它不会等待镜像服务器的确认。这意味着事务不需要等待镜像服务器将日志写入磁盘便可提交，但可能会丢失某些数据。

如果要实现自动故障转移功能的高安全性模式，则要求使用第 3 个服务器实例，称为“见证服务器”。见证服务器并不能用于数据库，而是通过验证主体服务器是否已启用并运行来支持自动故障转移。只有在镜像服务器和见证服务器与主体服务器断开连接之后而保持相互连接时，镜像服务器才启动自动故障转移。

将事务安全设置为 OFF 时，数据库镜像会话便会异步运行。异步操作仅支持一种操作模式：高性能模式。此模式可增强性能，但要牺牲高可用性。高性能模式仅使用主体服务器和镜像服务器。镜像服务器上出现的问题不会影响主体服务器，在丢失主服务器的情况下，镜像数据库将标记为 DISCONNECTED，但仍可以作为备用数据库。

高性能模式仅支持一种角色切换形式：强制服务（可能造成数据丢失），此服务使用

镜像服务器作为备用服务器。强制服务是可能对主体服务器故障作出的响应之一。由于可能造成数据丢失，因此，应当在将服务强制到镜像之前考虑其他备选服务器。

6.5.3　使用 T-SQL 配置数据库镜像

数据库镜像的配置相对于备份、恢复等数据库操作来说要复杂得多，为了便于读者理解，这里就去掉见证服务器，以最简单的两台数据库服务器：主服务器和镜像服务器的配置来进行讲解。

由于主服务器和镜像服务器都可以放在互联网上，所以主服务器和镜像服务器之间通信的安全性尤为重要。出于安全性的考虑，主服务器和镜像服务器之间的通信必须是可信任的，所以系统要求主服务器和镜像服务器最好是在同一个域中，通过域账户来进行验证。如果没有域，那么必须通过证书来验证相互之间的通信。

假设一个信用卡的数据库 Credit 需要配置数据库镜像，现在该数据库已经在服务器 A 上，所以就以 A 为主数据库，服务器 B 为镜像服务器，笔者当前的环境是：

- A 服务器：Windows 2003 SP2，SQL Server 2008，服务器名 ms-zy，IP 是 10.101.10.83；
- B 服务器：Windows 2003 SP2，SQL Server 2012，服务器名 ibm-pc，IP 是 10.101.10.86。

这两台服务器都在互联网上单独存在，并没有加入域。

技巧：读者若想学习数据库镜像配置而没有两台服务器，那么可以使用 Virtual PC 在虚拟机中安装 SQL Server，然后在虚拟机中配置。实际上，微软很多产品的实验环境都是在 Virtual PC 中搭建的。

配置镜像的具体操作如下所述。

（1）由于服务器并没有加入域，所以这里只能使用证书的方式。首先在 A 数据库上创建证书，关于证书已经在第 5 章的数据加密部分进行了介绍。此处创建证书的 SQL 脚本如代码 6.18 所示。

代码 6.18　在 A 数据库创建证书

```
USE master;
GO
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'your password' --创建主密钥
GO
CREATE CERTIFICATE MIR_A_cert --创建证书
   WITH SUBJECT = 'MIR_A certificate for database mirroring',
      start_date = '01/01/2013',
      EXPIRY_DATE = '10/31/2099' ;
GO
```

（2）根据创建的证书，为 A 数据库创建镜像端点，创建端点使用 CREATE ENDPOINT 命令，关于该命令的语法这里就不做详细介绍，读者可以通过联机丛书了解详细信息。创建镜像端点的 SQL 如代码 6.19 所示。

代码 6.19　使用证书为 A 数据库创建镜像端点

```
USE master;
```

```
GO
CREATE ENDPOINT Endpoint_Mirroring --镜像端点的名字
    STATE = STARTED
    AS TCP (
       LISTENER_PORT=5024            --镜像通信中所使用的端口
       , LISTENER_IP = ALL           --允许所有 IP
    )
    FOR DATABASE MIRRORING (
       AUTHENTICATION = CERTIFICATE MIR_A_cert --使用了第 1 步创建的证书
       , ENCRYPTION = REQUIRED ALGORITHM RC4
       , ROLE = ALL
    );
```

创建镜像端点后可以通过 SSMS 来查看，依次展开“服务器对象”、“端点”、Database Mirroring 便可看到当前服务器已有的镜像端点，如图 6.14 所示。

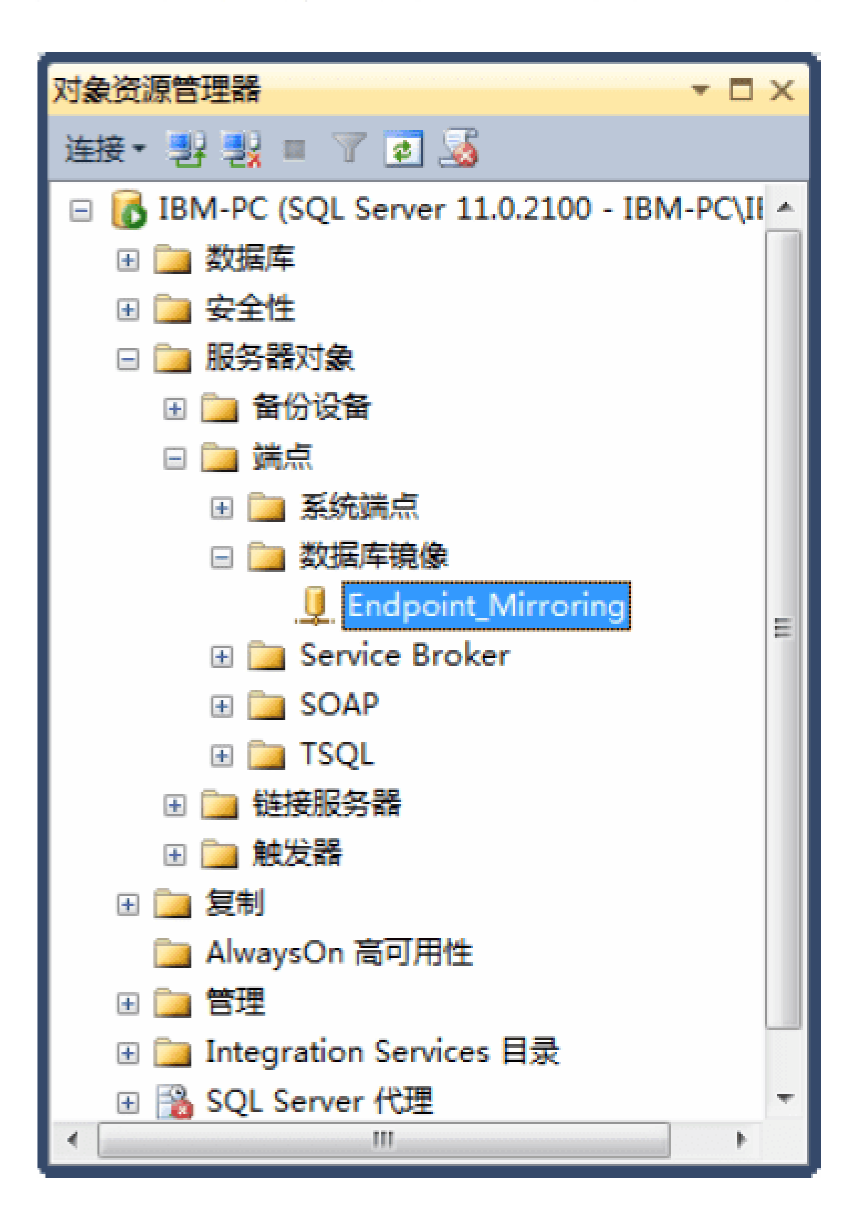


图 6.14　查看镜像端点

（3）将 A 服务器上创建的证书备份并复制到 B 服务器上，该证书就是 B 服务器与 A 服务器通信的有效凭证。备份证书使用 BACKUP CERTIFICATE 命令，具体 SQL 脚本如代码 6.20 所示。

代码 6.20　备份 A 服务器上创建的证书

```
USE master;
GO
BACKUP CERTIFICATE MIR_A_cert --备份证书
TO FILE = 'C:\MIR_A_cert.cer';
```

（4）使用同样的方法为 B 服务器创建证书 MIR_B_cert 并使用该证书的镜像端点 Endpoint_Mirroring，然后将 B 服务器上的证书备份并复制到 A 服务器上。具体 SQL 脚本如代码 6.21 所示。

代码 6.21　为 B 服务器创建证书、镜像端点并备份证书

```
USE master;
```

```
GO
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'your password' --创建主密钥
GO
CREATE CERTIFICATE MIR_B_cert                          --创建证书
   WITH SUBJECT = 'MIR_B certificate for database mirroring',
     start_date = '01/01/2013',
     EXPIRY_DATE = '10/31/2099' ;
GO
CREATE ENDPOINT Endpoint_Mirroring                     --镜像端点的名字
   STATE = STARTED
   AS TCP (
     LISTENER_PORT=5024                                --镜像通信中所使用的端口
     , LISTENER_IP = ALL                               --允许所有 IP
   )
   FOR DATABASE_MIRRORING (
     AUTHENTICATION = CERTIFICATE MIR_B_cert --使用前面创建的证书
     , ENCRYPTION = REQUIRED ALGORITHM RC4
     , ROLE = ALL
);
GO
BACKUP CERTIFICATE MIR_B_cert
TO FILE = 'C:\MIR_B_cert.cer';
```

（5）在 A 服务器上创建登录名和用户，该用户用于在镜像通信中连接 B 服务器。创建登录名和用户的 SQL 脚本如代码 6.22 所示。

代码 6.22　在 A 服务器上创建用于连接 B 服务器的登录名和用户

```
USE master;
GO
CREATE LOGIN MIR_B_login WITH PASSWORD = 'your password'; --创建登录名
GO
CREATE USER MIR_B_user FOR LOGIN MIR_B_login;             --创建用户
```

（6）将 B 服务器上创建的证书在 A 服务器还原，同时将证书的使用授予刚创建的用户，这样该用户便可在镜像中通过安全验证，与 B 服务器通信。还原证书并授予给用户的 SQL 脚本如代码 6.23 所示。

代码 6.23　在 A 服务器上还原证书

```
CREATE CERTIFICATE MIR_B_cert
   AUTHORIZATION MIR_B_user            --将用户账号与证书关联
   FROM FILE = 'C:\MIR_B_cert.cer'     --B 服务器上备份过来的证书
```

（7）使用 GRANT 命令将镜像端点连接的权限授予登录名。授权后 A 服务器同 B 服务器通信时便可使用该用户，而该用户又具有 B 服务器中的证书，从而保证了通信安全。授权的代码为：

```
GRANT CONNECT ON ENDPOINT::Endpoint_Mirroring TO [MIR_B_login];
```

（8）使用同样的方法在 B 服务器上创建登录名和用户，然后将 A 服务器上的证书还原到 B 服务器中并授予登录名对镜像端点的连接权限，具体操作 SQL 脚本如代码 6.24 所示。

代码 6.24　在 B 服务器上创建登录名、用户，并还原证书、授予权限

```
USE master;
GO
CREATE LOGIN MIR_A_login WITH PASSWORD = 'your password';
GO
CREATE USER MIR_A_user FOR LOGIN MIR_A_login
GO
CREATE CERTIFICATE MIR_A_cert --还原证书
    AUTHORIZATION MIR_A_user
    FROM FILE = 'C:\MIR_A_cert.cer';
GO
GRANT CONNECT ON ENDPOINT::Endpoint_Mirroring TO MIR_A_login
```

（9）将 A 服务器上的数据库完整备份并在 B 服务器上进行还原，以初始化镜像数据库，还原数据库时使用 NORECOVERY 模式。备份并还原数据库的 SQL 脚本如代码 6.25 所示。

代码 6.25　备份 A 服务器上的数据库并在 B 服务器还原

```
--A 服务器
BACKUP DATABASE Credit      --备份
TO DISK = 'C:\Credit.bak'
--B 服务器
RESTORE DATABASE Credit     --还原
    FROM DISK = 'C:\Credit.bak '
    WITH NORECOVERY
```

（10）在 B 服务器上配置 A 为镜像伙伴，在 A 服务器上配置 B 服务器为镜像伙伴。配置完成后 A 与 B 即完成了镜像功能，配置镜像伙伴的 SQL 脚本如代码 6.26 所示。

代码 6.26　配置镜像伙伴

```
--B 服务器
ALTER DATABASE Credit
    SET PARTNER = 'TCP://ms-zy:5024';   --设置镜像伙伴
--A 服务器
ALTER DATABASE Credit
    SET PARTNER = 'TCP://ms-zy2:5024';  --设置镜像伙伴
```

注意：必须先配置镜像服务器，然后再配置主服务器。

（11）镜像服务器已经配置完成，接下来用户可以连接到 A 服务器并提交数据更改，在镜像模式下 B 服务器的镜像数据库是无法访问的，只需在 A 服务器上运行代码 6.27 即可将角色进行互换。也就是说，角色互换后 A 服务器将作为镜像服务器，而 B 服务器作为主服务器，此时便可查看到对 A 服务器提交的数据更改已经通过数据库镜像功能同步到 B 服务器上。

代码 6.27　镜像角色互换

```
USE master;
ALTER DATABASE [Credit]
SET PARTNER FAILOVER --角色互换
```

通过以上步骤，已经在两台服务器上配置完成了数据库镜像，通过手动的镜像角色互

换可以将镜像中的主体服务器和镜像服务器进行互换。若要实现自动的镜像角色互换，那么就要使用第三台服务器作为见证服务器。见证服务器随时监控着另外两台服务器的状态，当主服务器宕机时便自动将主服务器的角色切换到镜像服务器上。

关于见证服务器的配置与前面的步骤无异，在没有加入域的情况下需要 3 台服务器之间相互交换证书来实现通信的安全。

6.5.4　使用 SSMS 配置数据库镜像

使用 SSMS 进行数据库镜像配置相对要简单一些，但是总体的配置思路与使用 T-SQL 没有差别，都是建立证书、交换证书、建立登录用户和建立镜像端点等操作。

在 SSMS 中配置数据库镜像的主要操作如下所述。

第（1）～（9）步与 6.5.3 节中的配置操作相同。

（10）在主服务器的对象资源管理器中，右击需要进行数据库镜像配置的数据库，在弹出的快捷菜单中选择“任务”选项下的“镜像”选项，系统打开镜像配置对话框，如图 6.15 所示。

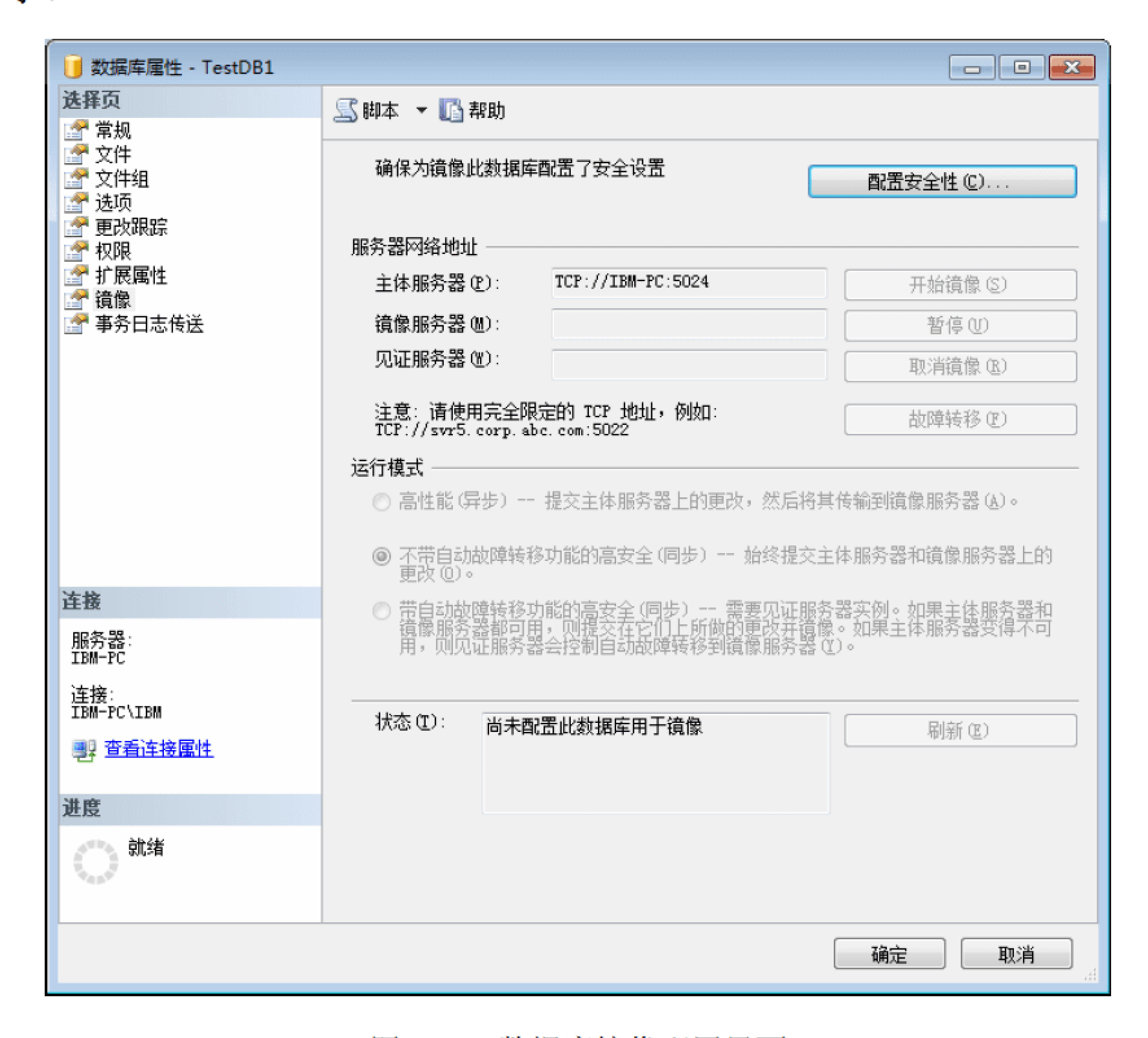


图 6.15　数据库镜像配置界面

说明：也可以右击该数据库，然后在弹出的快捷菜单中选择“属性”选项，在弹出的数据库属性窗口中选择“镜像”选项，同样也可以打开镜像配置窗口。

（11）单击“配置安全性”按钮，系统将打开“配置数据库镜像安全向导”对话框，直接单击“下一步”按钮，系统询问是否配置中包括见证服务器，如图 6.16 所示。

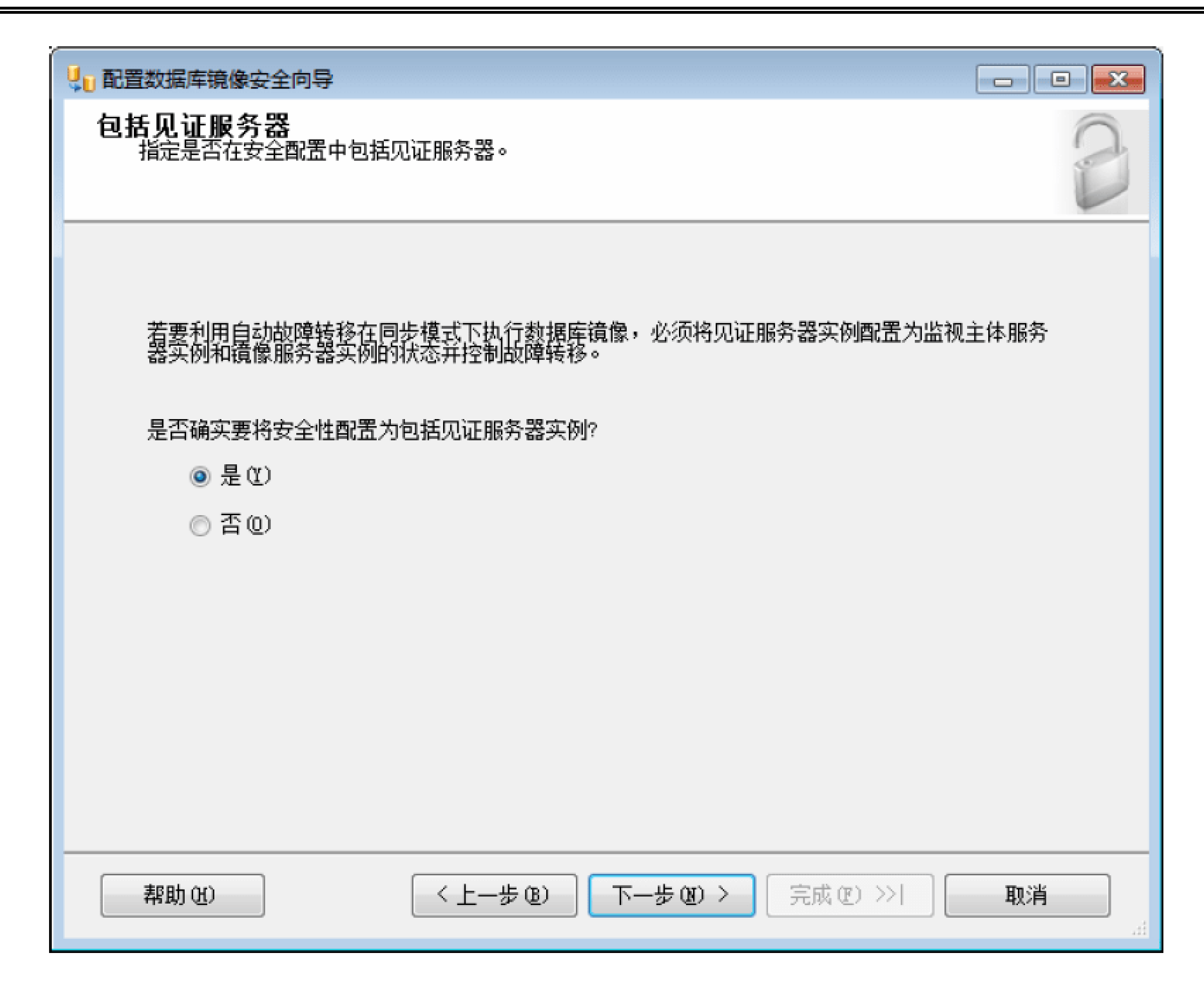


图 6.16　包括见证服务器选项

（12）此处不配置使用见证服务器，所以选择“否”单选框，然后单击“下一步”按钮，向导进入主体服务器配置界面，如图 6.17 所示。

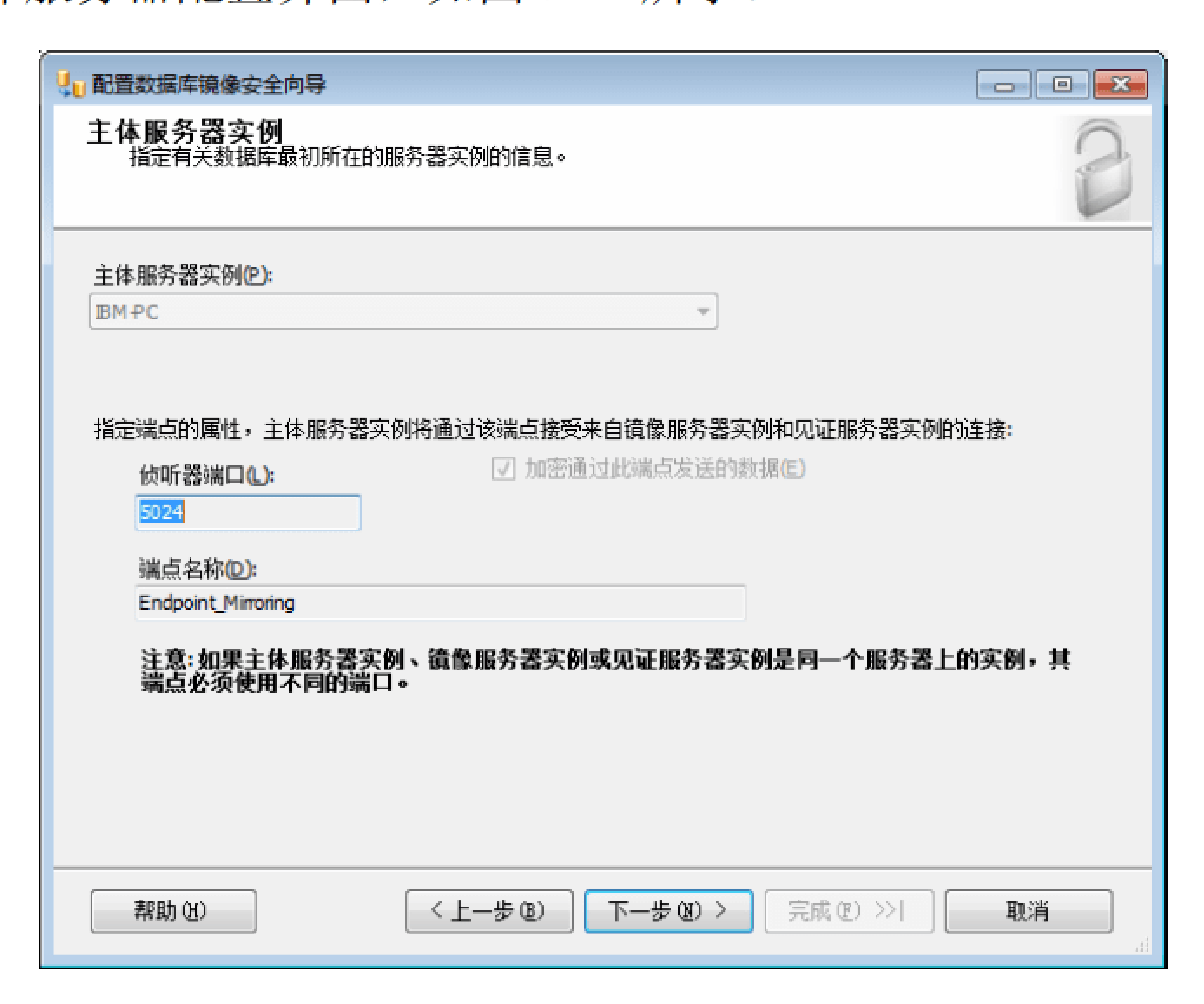


图 6.17　主体服务器实例配置

注意：这里也可以通过向导的方式来添加端点，但是由于向导中只能添加一般的端点，而不能添加使用带证书的端点。若是基于域认证的镜像配置，则可以直接使用向导来配置镜像端点。

（13）由于在前面的步骤中已经使用 T-SQL 建立了镜像端点，所以此处系统已经列出了端点名称。单击“下一步”按钮，向导进入镜像服务器实例配置界面，如图 6.18 所示。

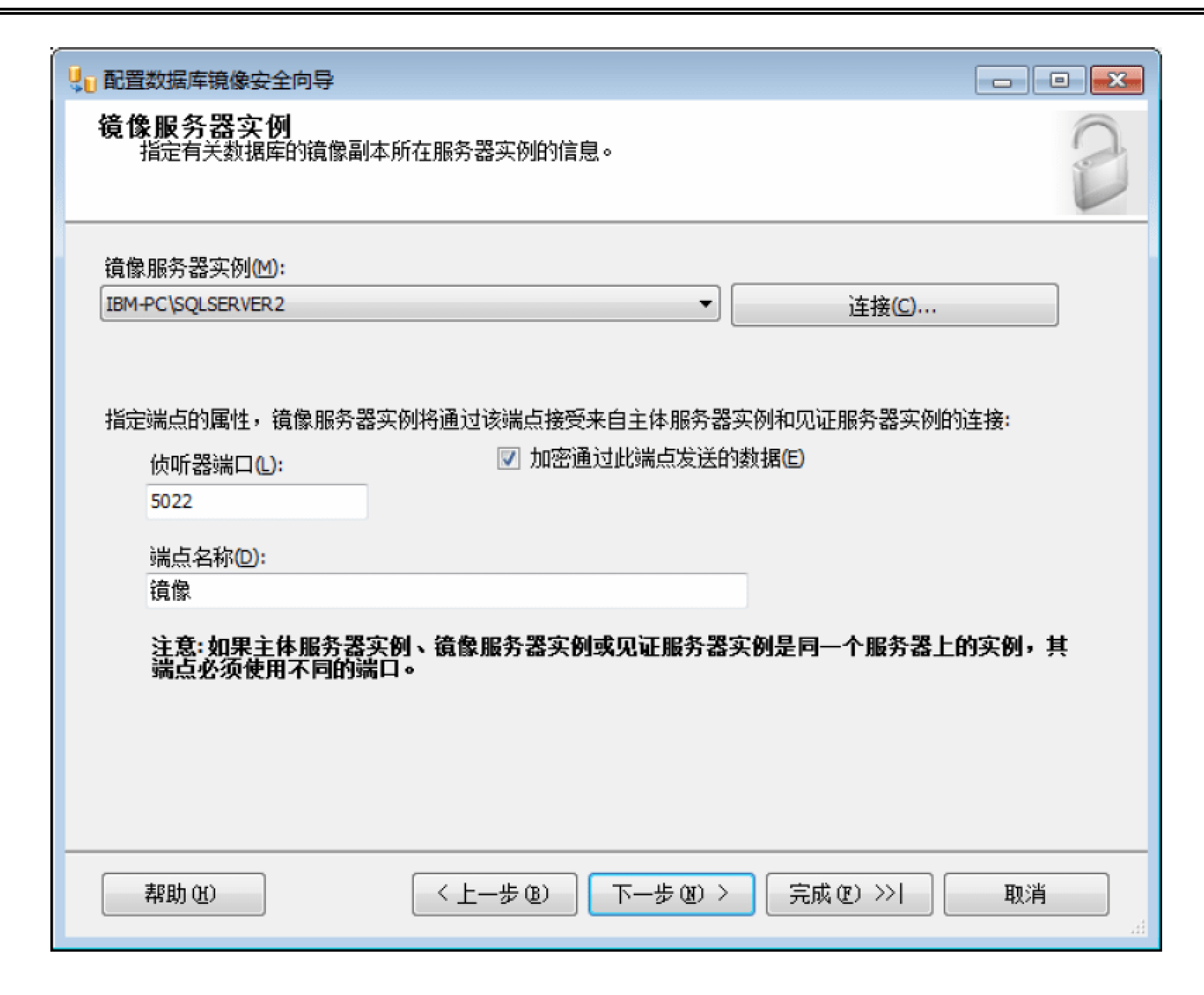


图 6.18　镜像服务器实例配置

（14）在镜像服务器实例配置界面中，单击“连接”按钮，连接到镜像服务器 IBM-PC\SERVER2，该服务器中也配置好了镜像端点，所以侦听端口和端点名称都已经显示出来。此处不用修改，直接单击“下一步”按钮，系统进入服务账号设置界面，如图 6.19 所示。

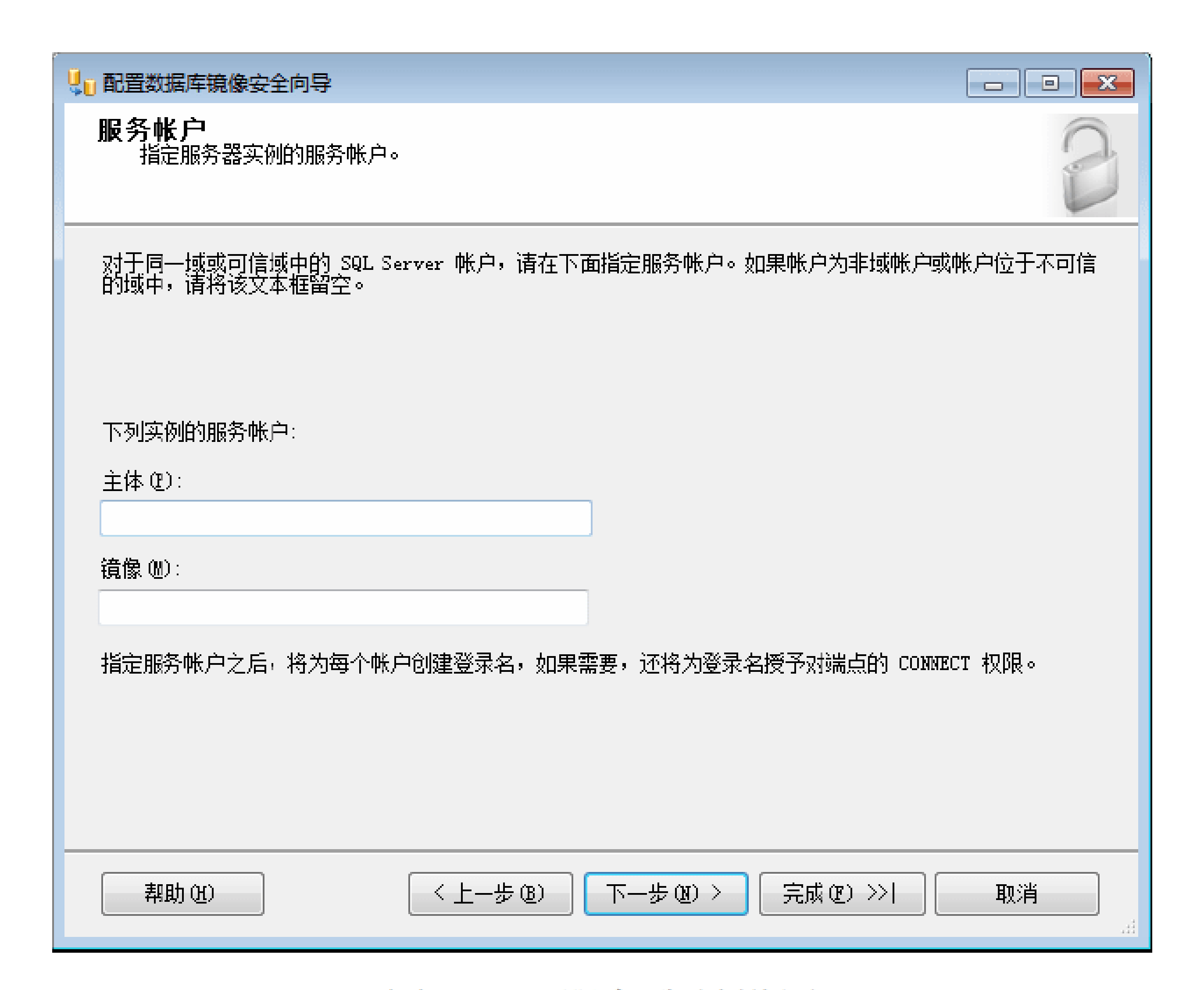


图 6.19　服务账号设置

（15）由于此处不是使用域账号进行通信，而是使用证书和对应的数据库用户进行镜像通信，所以此处不需要设置服务账户。单击“下一步”按钮，系统汇总显示要配置的服务器信息，最后单击“完成”按钮系统将完成对主体服务器和镜像服务器的配置。

（16）配置完成后，系统弹出对话框询问是否开始镜像，此处选“否”单选按钮，回到了数据库镜像配置主窗口，如图 6.20 所示。

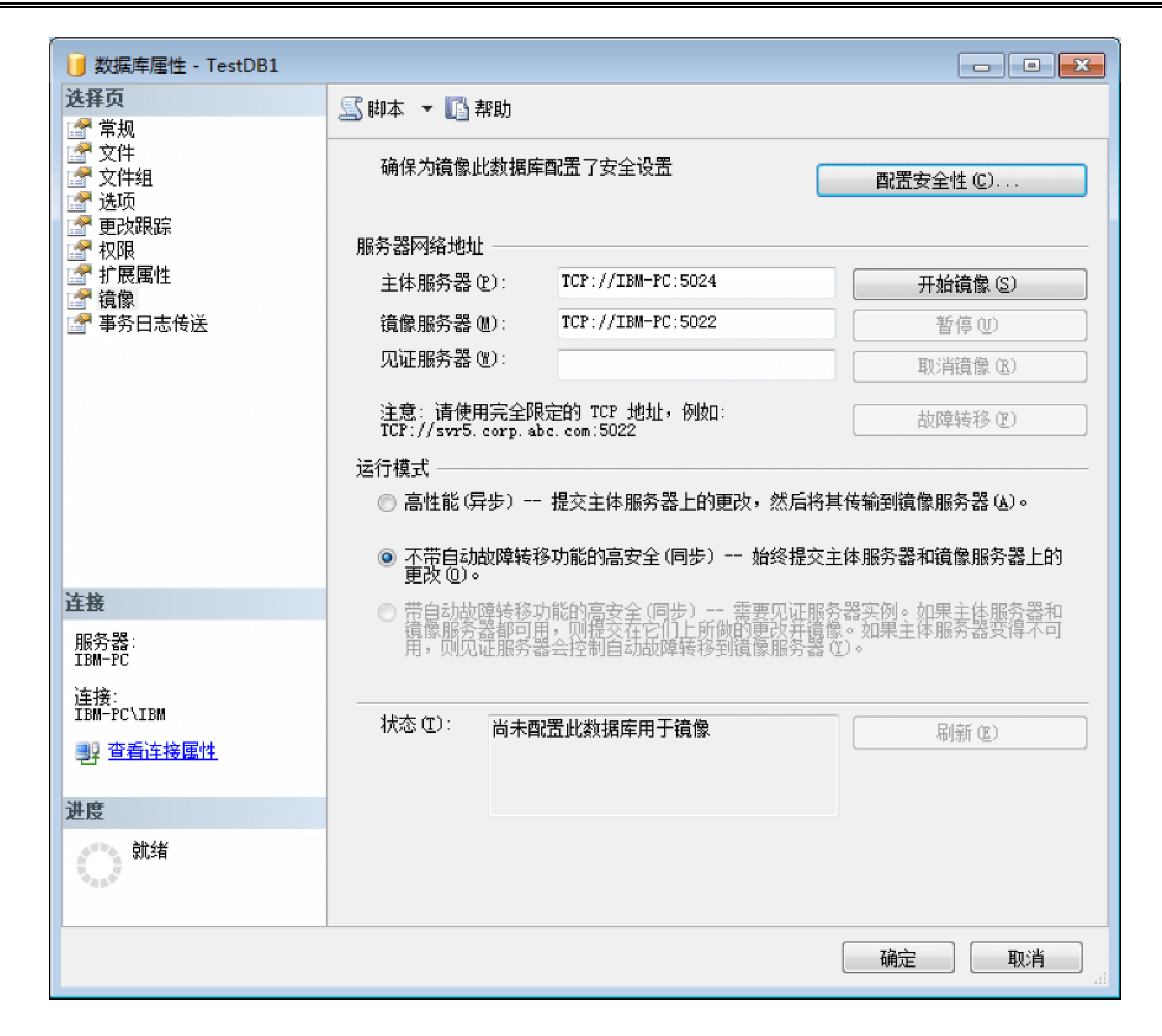


图 6.20　数据库镜像配置

（17）根据实际的需要，选择是使用高性能模式还是使用高安全模式，然后单击“开始镜像”按钮，系统正式进入镜像状态。

（18）单击“确定”按钮，完成数据库镜像的配置，在主体服务器的 SSMS 中可以看到该数据库旁有“（主体，已同步）”字样，说明数据库镜像配置成功，并且已经处于正常运行状态。

6.6　日 志 传 送

除了数据库镜像技术外，还可以通过日志传送的方式来提高数据的安全性和系统的可用性。日志传送和高可用模式下的数据库镜像类似，是一种常用的数据库温备技术。本机就主要讲解日志传送的概念和配置。

6.6.1　日志传送概述

日志传送可以自动将主服务器实例上指定数据库内的事务日志备份，发送到另外的一个或多个辅助服务器实例上，然后每个辅助服务器上将还原接收到的日志并应用于辅助数据库中。日志传送中也可选第三个服务器实例（称为“监视服务器”）来记录备份和还原操作的历史记录及状态。监视服务器还可以在无法按计划执行这些操作时引发警报。日志

传送由以下 3 项操作组成：

- 在主服务器实例中备份事务日志。
- 将事务日志文件复制到辅助服务器实例。
- 在辅助服务器实例中还原日志备份。

日志可传送到多个辅助服务器实例。在这些情况下，将针对每个辅助服务器实例重复执行复制操作和日志还原操作。

注意：日志传送配置不会自动从主服务器故障转移到辅助服务器。如果主数据库变为不可用，可手动使任意辅助数据库联机。

6.6.2　日志传送的服务器角色

在日志传送中有主服务器、辅助服务器和监视服务器 3 个角色。这 3 个角色在日志传送当中负责不同的工作。

1．主服务器和数据库

与数据库镜像中的定义一样，日志传送配置中的主服务器是作为生产服务器的 SQL Server 数据库引擎实例。主数据库是主服务器上需要进行日志传送到其他服务器的数据库。通过 SSMS 进行的所有日志传送配置管理都是在主数据库中执行的。

注意：主数据库必须使用完整恢复模式或大容量日志恢复模式，将数据库切换为简单恢复模式会导致日志传送停止工作。

2．辅助服务器和数据库

日志传送配置中的辅助服务器是用于还原日志在其中保留主数据库备用副本的服务器。不仅一台主服务器可以配置多台辅助服务器，一台辅助服务器也可以包含多台不同主服务器中数据库的备份副本。在一台辅助服务器上如果配置有多个主数据库副本，为了应对多个主系统同时不可用的罕见情况，辅助服务器的规格可以比各主服务器高。

辅助数据库必须通过还原主数据库的完整备份的方法进行初始化。还原时可以使用 NORECOVERY 或 STANDBY 选项。

3．监视服务器

日志传送中监视服务器是可选的，它并不能像数据库镜像那样进行自动的故障转移，但可以跟踪日志传送的所有细节，包括：

- 主数据库中事务日志最近一次备份的时间。
- 辅助服务器最近一次复制和还原备份文件的时间。
- 有关任何备份失败警报的信息。

监视服务器应独立于主服务器和辅助服务器，否则由于主服务器或辅助服务器的宕机，如果监视服务器也在同一台机器上，则会丢失关键信息和中断监视。一台监视服务器可以监视多个日志传送配置。在这种情况下，使用该监视服务器的所有日志传送配置将共

享一个警报作业。

6.6.3　日志传送的定时作业

日志传送是以 SQL Server 作业的方式定时执行，涉及 4 项由专用 SQL Server 代理作业处理的作业。这些作业包括备份作业、复制作业、还原作业和警报作业。

1．备份作业

备份作业是在主服务器实例上运行，主要负责为每个主数据库执行备份操作，将历史记录信息记录到本地服务器和监视服务器上，并删除旧备份文件和历史记录信息。默认情况下，备份作业每 2 分钟执行一次，但是间隔是可自定义的。

启用日志传送后，将在主服务器实例上创建 SQL Server 代理作业类别“日志传送备份”。

SQL Server 2012 企业版及更高版本支持备份压缩。是否压缩给定日志备份取决于 backup compression default 服务器配置选项的当前设置。

2．复制作业

复制作业是在每个辅助服务器实例上创建的。此作业将备份文件从主服务器复制到辅助服务器中的可配置目标，并在辅助服务器和监视服务器中记录历史记录。复制作业计划应与备份计划相似，也可自定义。

启用日志传送后，将在辅助服务器实例上创建 SQL Server 代理作业类别“日志传送复制”。

3．还原作业

还原作业是在辅助服务器实例上为每个日志传送配置创建一个作业。也就是说，有多个日志传送将会有多个还原作业。此作业将复制的备份文件还原到辅助数据库，同时将历史记录信息记录在本地服务器和监视服务器上，并删除旧文件和旧历史记录信息。

在启用日志传送时，辅助服务器实例上会创建 SQL Server 代理作业类别“日志传送还原”。

还原作业的执行频率可以按照复制作业的频率计划，也可以延迟还原作业。使用相同的频率计划这些作业可以使辅助数据库尽可能与主数据库保持紧密一致，便于创建备用数据库。相反，如果延迟还原作业，那么在主数据库出现严重的用户错误（如删除表或不适当地删除表行）情况下是很有用的。如果知道出错的时间，则可以将该辅助数据库向前移动到错误发生生前，然后就可以导出丢失的数据将其导回到主数据库。

4．警报作业

如果使用了监视服务器，警报作业将在警报监视器服务器实例上创建。此警报作业由使用监视器服务器实例的所有日志传送配置中的主数据库和辅助数据库所共享。警报作业在监视服务器上只创建一个，并不会为多个监视的日志传送服务创建多个警报。对警报作业进行的任何更改（例如，重新计划作业、禁用作业或启用作业）会影响所有使用监视服

务器的数据库。如果在指定的阈值内未能成功完成备份和还原操作，警报作业将引发主数据库和辅助数据库警报（您必须指定警报编号）。在警报作业中必须为这些警报配置一个操作员来接收日志传送失败的通知。

在启用日志传送时，监视服务器实例上会创建 SQL Server 代理作业类别“日志传送警报”。

如果未使用监视服务器，系统将在主服务器实例和每个辅助服务器实例上分别创建一个警报作业。如果在指定的阈值内未能成功完成备份操作，主服务器实例上的警报作业将引发错误。如果在指定的阈值内未能成功完成本地复制和还原操作，辅助服务器实例上的警报作业将引发错误。

6.6.4　使用 T-SQL 配置日志传送

日志传送主要基于 SQL Server 代理，使用定时作业来完成，另外在配置日志传送之前必须要创建共享文件夹，用于辅助服务器访问。这里假设有数据库 logTrans1 需要进行日志传送，共享文件夹为“C:\data”，在 T-SQL 中配置日志传送主要有以下几步操作。

（1）备份主数据库并在辅助服务器上还原主数据库的完整备份，初始化辅助数据库。具体操作如代码 6.28 所示。

代码 6.28　备份和还原数据库

```
backup database logTrans1  --在主数据库上备份
to disk='c:\logt.bak'
--以下是将数据库还原到辅助数据库上
restore database logTrans2
from disk='c:\logt.bak'
with NORECOVERY,
move 'logTrans' to 'c:\logTrans2.mdf',
move 'logTrans_log' to 'c:\logTrans2.ldf'
```

（2）在主服务器上，执行 sp_add_log_shipping_primary_database 以添加主数据库。存储过程将返回备份作业 ID 和主 ID。具体 SQL 脚本如代码 6.29 所示。

代码 6.29　配置日志传送主数据库

```
DECLARE @LS BackupJobId AS uniqueidentifier
DECLARE @LS PrimaryId AS uniqueidentifier
--配置主数据库
EXEC master.dbo.sp_add_log_shipping_primary_database 配置主数据库
@database = N'logTrans1'
,@backup directory = N'D:\data'
,@backup share = N'\\10.101.10.66\data'
,@backup job name = N'LSBackup logTrans1'
,@backup retention period = 1440
,@monitor server = N'localhost'
,@monitor server security mode = 1
,@backup threshold = 60
,@threshold alert enabled = 0
,@history retention period = 1440
,@backup job id = @LS BackupJobId OUTPUT
,@primary_id = @LS_PrimaryId OUTPUT
```

```
,@overwrite = 1
```

（3）在主服务器上，执行 sp_add_jobschedule 以添加使用备份作业的计划。为了能够尽快看到日志传送的效果，这里将日志备份的频率设置为 2 分钟一次。但是在实际生产环境中，一般是用不到这么高的执行频率的。添加计划的脚本如代码 6.30 所示。

注意：sp_add_jobschedule 存储过程是在 msdb 数据库中，在其他数据库中是没有该存储过程的。

代码 6.30　添加备份计划

```
DECLARE @schedule_id int
EXEC msdb.dbo.sp_add_jobschedule @job_name =N'LSBackup_logTrans1',
--SQL 作业计划
@name=N'BackupDBEvery2Min',
@enabled=1,
@freq_type=4,
@freq_interval=1,
@freq_subday_type=4,
@freq_subday_interval=2,
@freq_relative_interval=0,
@freq_recurrence_factor=1,
@active_start_date=20080622,
@active_end_date=99991231,
@active_start_time=0,
@active_end_time=235959,
@schedule_id = @schedule_id OUTPUT
select @schedule_id
```

（4）在监视服务器上，执行 sp_add_log_shipping_alert_job 以添加警报作业。此存储过程用于检查是否已在此服务器上创建了警报作业。如果警报作业不存在，此存储过程将创建警报作业并将其作业 ID 添加到 log_shipping_monitor_alert 表中。在默认情况下，将启用警报作业并按计划每 2 分钟运行一次。添加警报作业脚本如代码 6.31 所示。

代码 6.31　添加警报作业

```
USE master
GO
EXEC sp_add_log_shipping_alert_job;
```

（5）在主服务器上，启用备份作业。启用作业使用 sp_update_job 存储过程，只需要输入作业名并设置状态为 1 即可。具体 SQL 脚本如代码 6.32 所示。

代码 6.32　启用备份作业

```
EXEC msdb.dbo.sp_update_job
@job_name='LSBackup_logTrans1',
@enabled=1
```

（6）在辅助服务器上，执行 sp_add_log_shipping_secondary_primary，提供主服务器和数据库的详细信息。此存储过程返回辅助 ID 以及复制和还原作业 ID。具体 SQL 脚本如代码 6.33 所示。

代码 6.33　设置复制和还原作业

```
DECLARE @LS_Secondary__CopyJobId uniqueidentifier
DECLARE @LS_Secondary__RestoreJobId uniqueidentifier
DECLARE @LS_Secondary__SecondaryId uniqueidentifier
EXEC master.dbo.sp add log shipping secondary primary  --设置复制和还原作业
@primary server = N'10.101.10.66'
,@primary database = N'logTrans1'
,@backup source directory = N'\\10.101.10.66\data'
,@backup destination directory = N'D:\log'
,@copy job name = N'LSCopy logTrans1'
,@restore job name = N'LSRestore logTrans2'
,@file retention period = 1440
,@copy_job_id = @LS_Secondary__CopyJobId OUTPUT
,@restore_job_id = @LS_Secondary__RestoreJobId OUTPUT
,@secondary_id = @LS_Secondary__SecondaryId OUTPUT
```

（7）在辅助服务器上，执行 sp_add_jobschedule 以设置复制和还原作业的计划。这里一般将复制和还原作业计划的频率设置为和日志备份的作业频率相同，所以此处将这两个作业的频率设置为每 2 分钟执行一次。具体 SQL 脚本如代码 6.34 和代码 6.35 所示。

代码 6.34　设置复制作业的计划

```
DECLARE @schedule_id int
--设置复制作业计划
EXEC msdb.dbo.sp add jobschedule
@job name=N'LSCopy logTrans1',
@name=N'CopyEvery2Min',
@enabled=1,
@freq type=4,
@freq interval=1,
@freq subday type=4,
@freq_subday_interval=2,
@freq relative interval=0,
@freq recurrence factor=1,
@active_start_date=20080622,
@active_end_date=99991231,
@active_start_time=0,
@active_end_time=235959,
@schedule_id = @schedule_id OUTPUT
select @schedule_id
```

代码 6.35　设置还原作业的计划

```
DECLARE @schedule id int
EXEC msdb.dbo.sp_add_jobschedule --设置还原作业的计划
@job name=N'LSCopy logTrans1',
@name=N'RestoreEvery2Min',
@enabled=1,
@freq type=4,
@freq interval=1,
@freq subday type=4,
@freq subday interval=2,
@freq relative interval=0,
@freq recurrence factor=1,
@active start date=20080622,
@active_end_date=99991231,
```

```
@active start time=0,
@active end time=235959,
@schedule id = @schedule id OUTPUT
select @schedule_id
```

（8）在辅助服务器上，执行 sp_add_log_shipping_secondary_database 以添加辅助数据库。具体操作脚本如代码 6.36 所示。

代码 6.36　添加辅助数据库

```
EXEC master.dbo.sp_add_log_shipping_secondary_database --添加辅助数据库
@secondary_database = N'logTrans2'
,@primary_server = N'10.101.10.66'
,@primary database = N'logTrans1'
,@restore delay = 0
,@restore mode = 1
,@disconnect_users = 0
,@restore threshold = 45
,@threshold alert enabled = 0
,@history_retention_period = 1440
GO
```

（9）在主服务器上，执行 sp_add_log_shipping_primary_secondary 向主服务器添加有关新辅助数据库的必需信息。具体 SQL 脚本如代码 6.37 所示。

代码 6.37　向主服务器添加辅助数据库的必需信息

```
EXEC master.dbo.sp_add_log_shipping_primary_secondary
@primary_database = N'logTrans1'
, @secondary_server = N'10.101.10.67' --辅助数据库的 IP
, @secondary_database = N'logTrans2'
```

（10）在辅助服务器上，启用复制和还原作业。启用作业仍然使用 sp_update_job 存储过程。具体操作如代码 6.38 所示。

代码 6.38　启用复制和还原作业

```
EXEC msdb.dbo.sp_update_job --启用复制作业
@job name='LSCopy logTrans1',
@enabled=1
EXEC msdb.dbo.sp_update_job --启用还原作业
@job name='LSRestore logTrans2',
@enabled=1
```

通过以上 10 步操作就完成了对日志传送的配置。现在每隔 2 分钟，系统将会把主服务器中的日志备份到共享文件夹中，辅助服务器访问共享文件夹将日志备份复制到本地硬盘上，然后由还原作业将复制到本地的日志还原到数据库，从而完成了日志的传送。用户可以在共享文件夹和辅助服务器的本地复制文件夹中看到备份的日志文件。

说明：在 SSMS 中可以通过右击对应的作业，在弹出的快捷菜单中选择“查看历史记录”选项来查看该作业是否正常运行。如果所有日志传送正常运行，则说明日志传送正常。

6.6.5　使用 SSMS 配置日志传送

结合前面介绍的各个技术点，本节学习使用 SSMS 配置日志传送的详细步骤。

（1）将主服务器的数据库 test1 备份，然后将数据库在辅助服务器上还原为数据库 test2。

（2）在主服务器 SSMS 的对象资源管理器中，右击要进行日志传送的数据库。在弹出的快捷菜单中选择“任务”选项下的“传送事务日志”命令，系统打开“数据库属性”对话框，如图 6.21 所示。

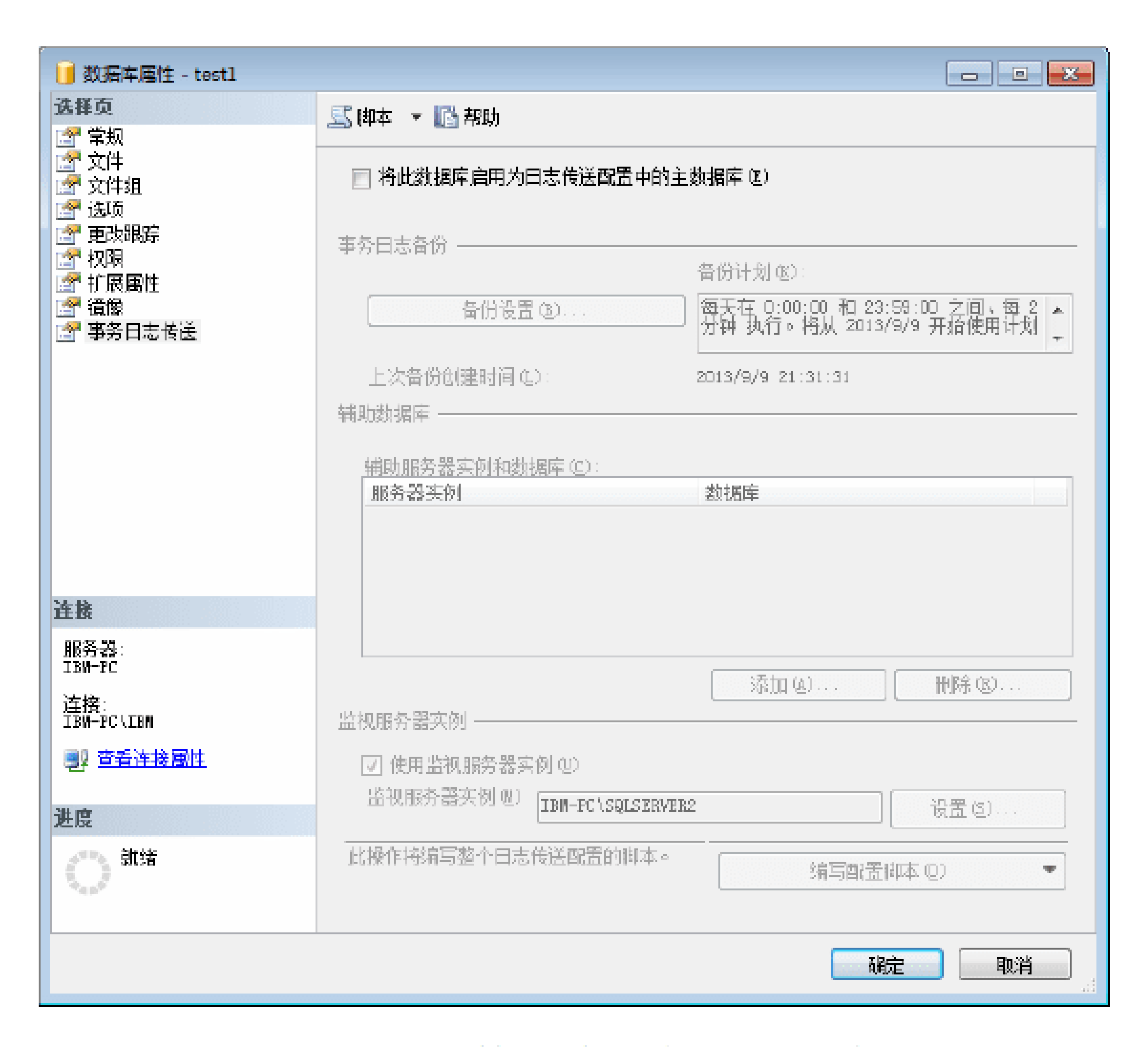


图 6.21　“数据库属性”对话框

（3）选中“将此数据库启用为日志传送配置中的主数据库”复选框，“备份设置”按钮变成可用状态。

（4）单击“备份设置”按钮，系统将弹出“事务日志备份设置”对话框，如图 6.22 所示。



图 6.22　“事务日志备份设置”对话框

（5）在“备份文件夹的网络路径”文本框中输入共享文件夹“\\10.101.10.66\data”，由于备份文件夹位于主服务器上，所以还需要输入备份文件夹的本地路径，这里是 D:\data。

（6）单击“计划”按钮，系统弹出“新建作业计划”对话框，如图 6.23 所示。

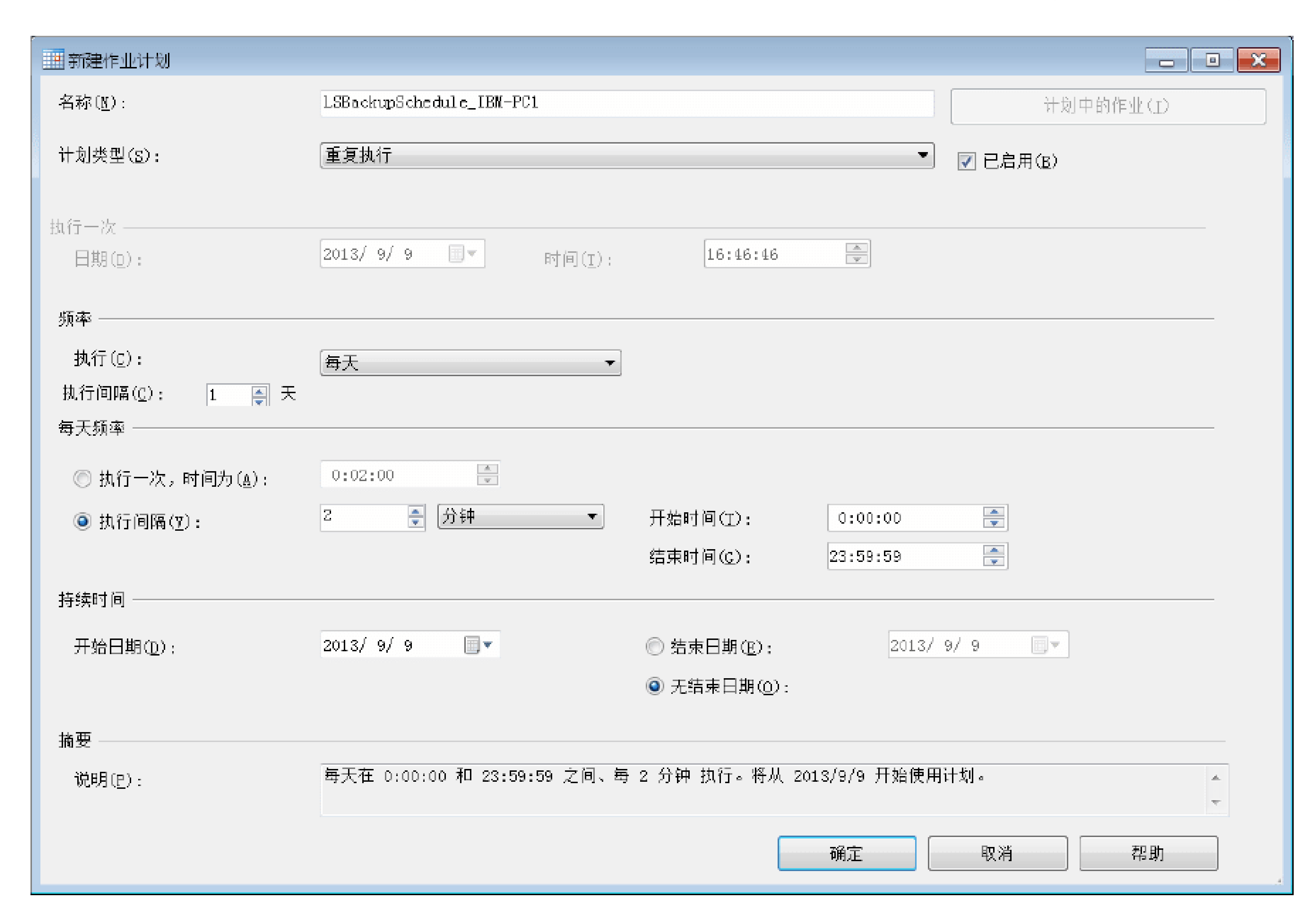


图 6.23　“新建作业计划”对话框

（7）设置作业执行的频率为每天执行，执行间隔为 2 分钟，然后单击“确定”按钮系统回到“数据库属性”对话框。

（8）接下来就是添加辅助数据库，单击“添加”按钮，系统打开“辅助数据库设置”对话框，如图 6.24 所示。

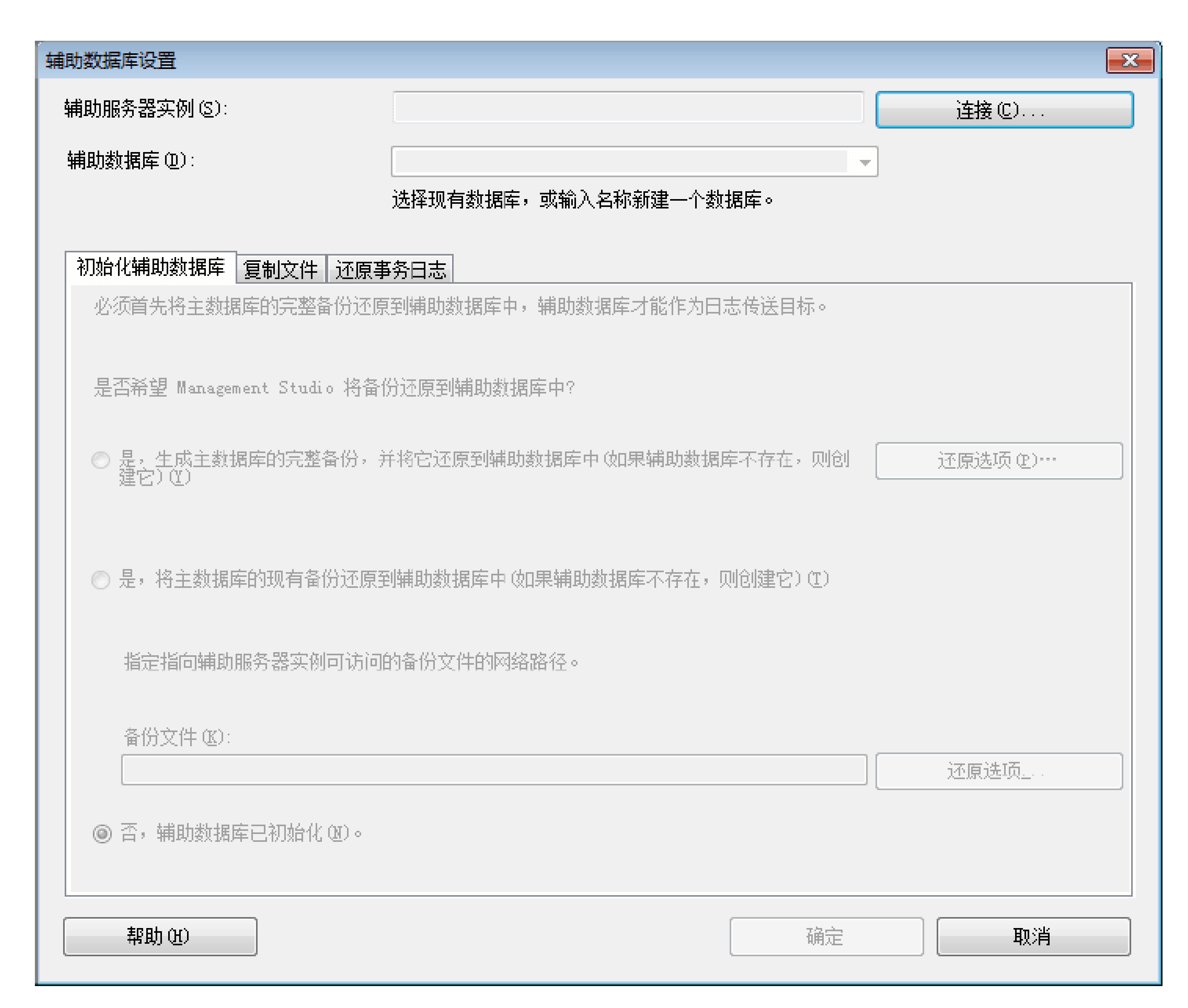


图 6.24　“辅助数据库设置”对话框

（9）单击“连接”按钮，连接到辅助数据库，并选择辅助数据库为 test2。由于已经将数据库还原到辅助数据库中，所以在“初始化辅助数据库”选项卡中选择“否，辅助数据库已初始化”单选按钮。

（10）切换到“复制文件”选项卡，输入日志备份要复制到辅助数据库的具体位置，这里设置辅助数据库的 C:\log 文件夹为日志存放的文件夹。然后单击“计划”按钮，设置每 2 分钟执行一次复制操作。设置完后如图 6.25 所示。



图 6.25　复制设置

（11）切换到“还原事务日志”选项卡，通过使用辅助服务器进行只读查询处理，可以减少主服务器的负荷。辅助数据库必须处于 STANDBY 模式才能执行此操作。如果数据库处于 NORECOVERY 模式，则不能运行查询。

使辅助数据库处于备用模式时，有两种配置方式：

- 还原事务日志备份时，可以选择数据库用户断开连接。如果选中此选项，则日志传送还原作业每次尝试将事务日志还原到辅助数据库时，用户都将与数据库断开连接。断开连接将按照还原作业设置的计划发生。
- 可以选择不与用户断开连接。在这种情况下，如果用户连接到辅助数据库，则还原作业无法将事务日志备份还原到辅助数据库。事务日志备份将一直累积到没有用户连接到该数据库为止。

如果希望在辅助数据库中能够进行只读查询，则此处选择“备用模式”单选框。单击“计划”按钮，设置还原事务日志的计划为 2 分钟执行一次，设置后如图 6.26 所示。

（12）单击“确定”按钮，完成辅助数据库的设置，系统回到“数据库属性”对话框。若要配置监视服务器，可以选中“使用监视服务器实例”复选框，然后单击“设置”按钮，

系统弹出“日志传送监视器设置”对话框，如图 6.27 所示。

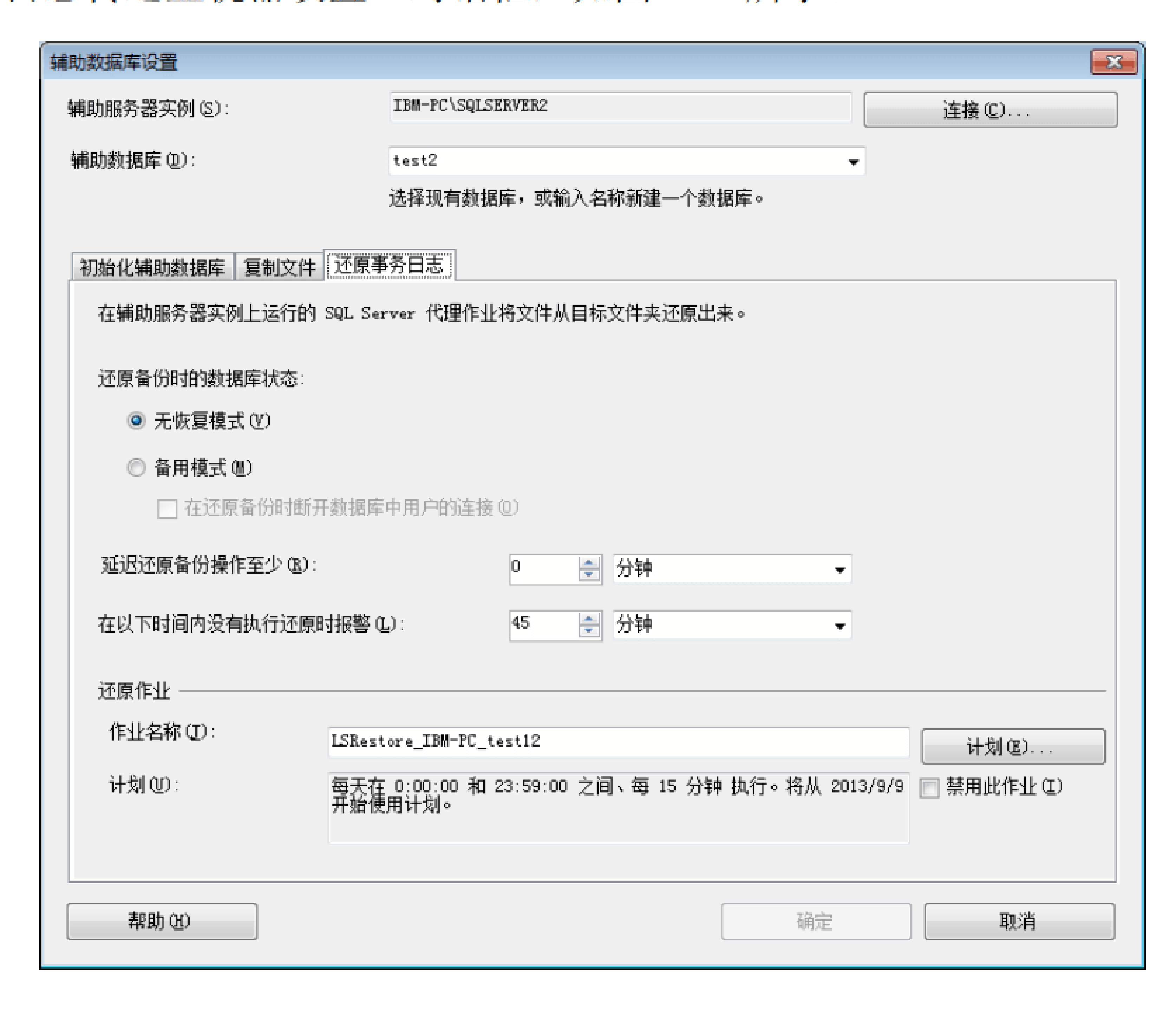


图 6.26　还原事务日志设置

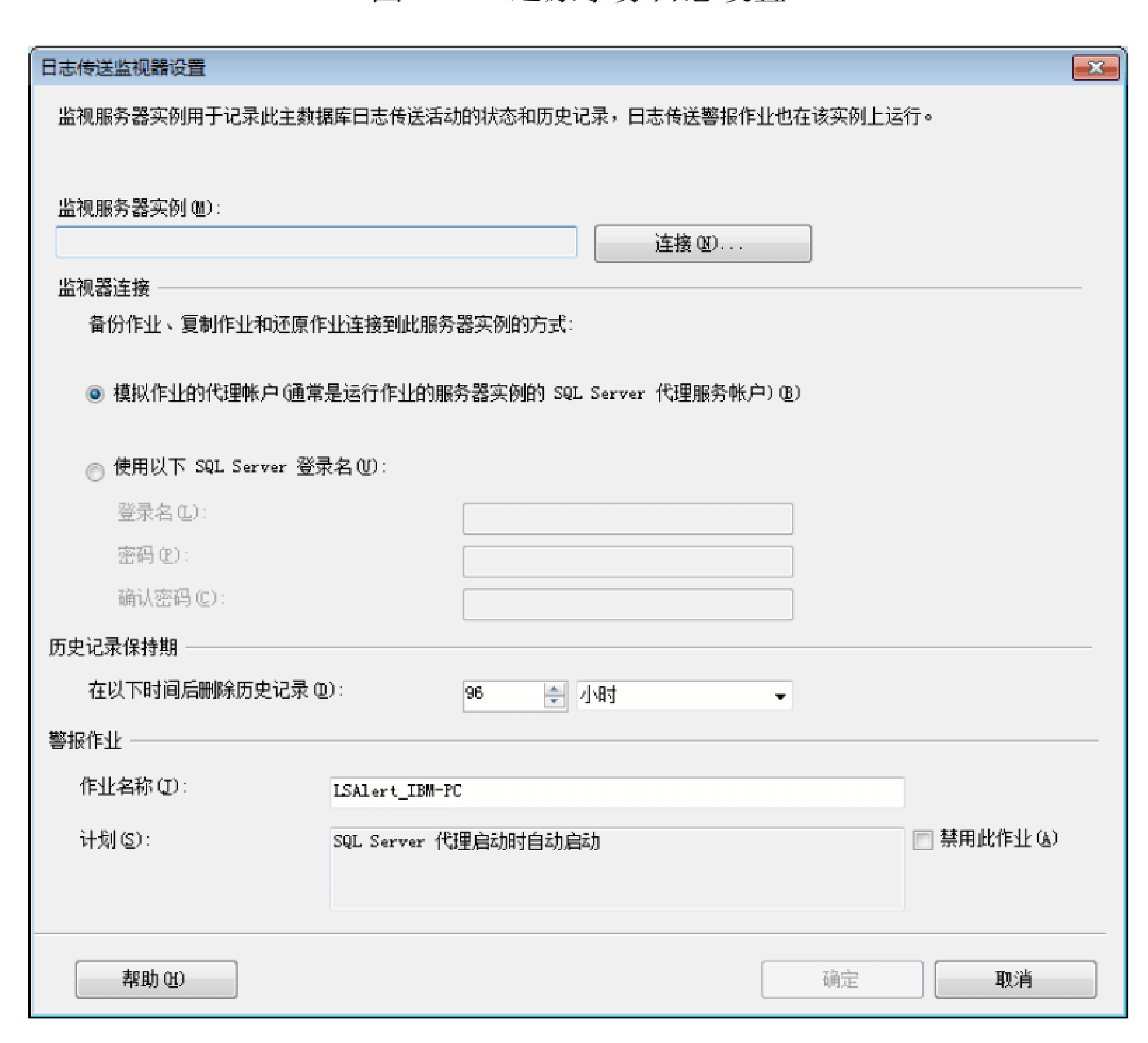


图 6.27　“日志传送监视器设置”对话框

（13）单击“连接”按钮，连接到监视服务器，然后单击“确定”按钮即可完成监视服务器的设置，系统回到“数据库属性”对话框。

（14）在其中单击“确定”按钮，系统正式启动事务日志的传送，界面如图 6.28 所示。

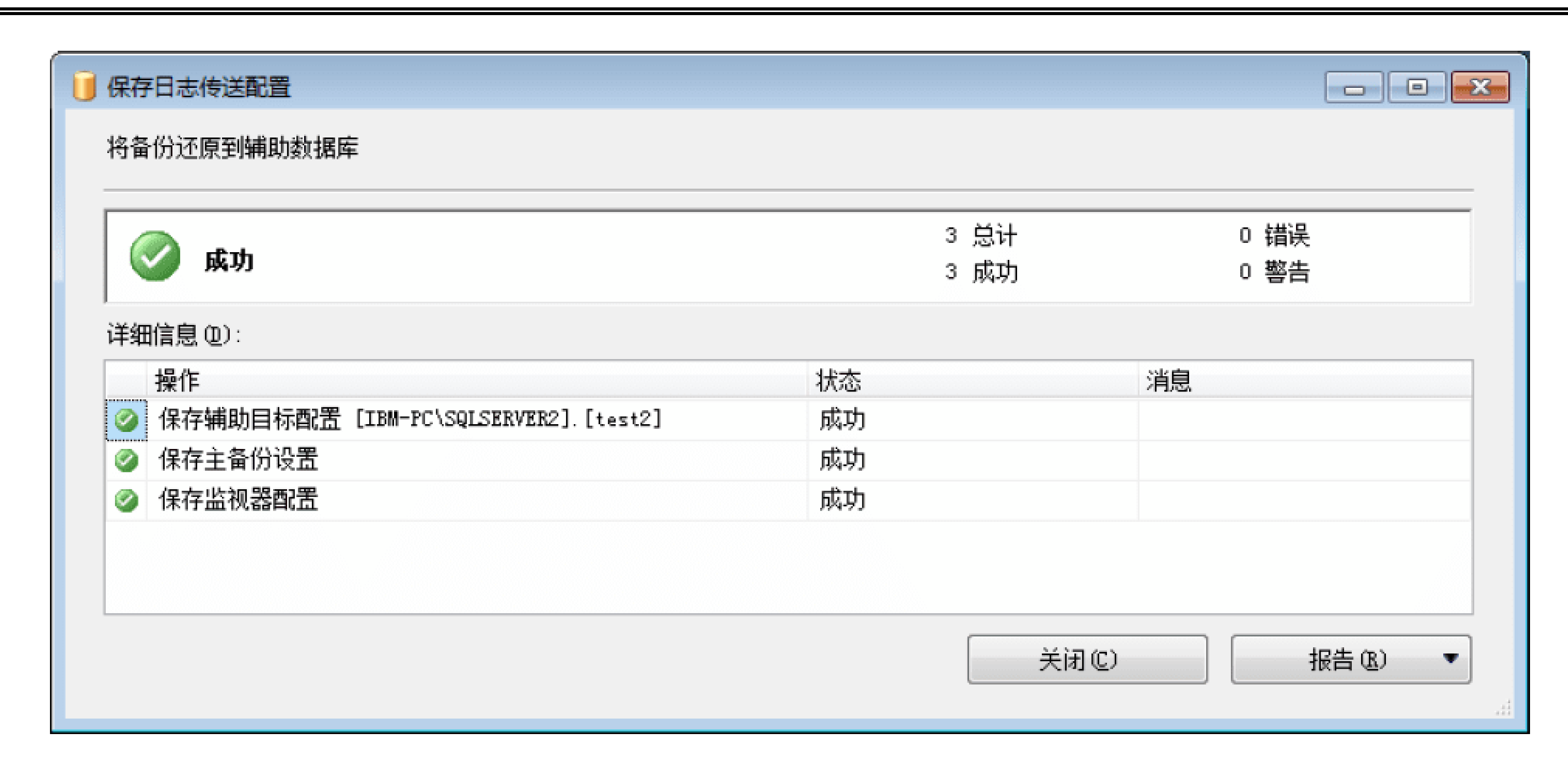


图 6.28　启动事务日志传送

6.7　数据库群集

故障转移群集是数据库热备技术中最安全的一种高可用性方案，但是故障转移群集也是配置最复杂，要求最高的解决方案。由于故障转移群集的配置十分复杂，而且需要相应的硬件环境支持，所以本节只讲解数据库的故障转移群集的基本知识。

6.7.1　群集简介

计算机群集（Cluster）的出现和使用已经有十几年的历史。作为最早的群集技术设计师之一，G.Pfister 对群集的定义是：“一种并行或分布式的系统，由全面互连的计算机集合组成，可作为一个统一的计算资源使用”。

在一个计算机群集中，内部是由多台服务器组合而成，对外来说用户或管理员不必了解群集的细节，不用关心具体访问了哪台服务器，也不用分担计算负载。如果其中的某台计算机发生故障，其他计算机将顶替发生故障的计算机，继续提供服务。例如，如果服务器群集中的任何资源发生了故障，则不论发生故障的组件是硬件还是软件资源，作为一个整体的群集都可以使用群集中其他服务器上的资源来继续向用户提供服务，而对客户端来说提供的服务以及服务的接口并没有变化。

当群集中的某资源发生故障时，由于需要资源的接管，所以同服务器群集连接的用户可能经历短暂的性能下降现象，但不会完全失去对服务的访问能力。如果需要提高整个群集的处理能力时，可以通过滚动升级过程来添加新资源或更换软硬件。该过程中，群集在整体上将保持联机状态，它不仅可供用户使用，而且在升级后，其性能也将得到改善。

群集服务（Cluster service）的优点包括以下几方面。

- 高可用性：通过服务器群集，资源（例如，磁盘驱动器和 IP 地址）的所有权会自动从故障服务器转移到可用的服务器。当群集中的某个系统或应用程序发生故障时，群集软件会在可用的服务器上重新启动故障应用程序，或者将工作从故障节

点分散到剩下的节点上。由此，用户只在瞬间内能感觉到服务的暂停。

- 故障恢复：当故障服务器重新回到其预定的首选所有者的联机状态时，群集服务将自动在群集中重新分配工作负荷。该特性可配置，但默认禁用。
- 可管理性：可以使用“群集管理器”工具（CluAdmin.exe），将群集作为一个单一的系统进行管理，并对犹如运行于一个单一服务器的应用程序实施管理，也可以将应用程序转移到群集中的其他服务器。“群集管理器”可用于手动平衡服务器的工作负荷，并针对计划维护释放服务器，还可以监控群集的状态、所有节点以及来自网络任何地方的资源。
- 可伸缩性：群集服务可扩展以满足需求的增长。当群集监督应用程序的总体负荷超出了群集的能力范围时，可以添加附加的节点。

服务器群集功能需要在企业版的服务器操作系统中才提供，也就是说这一功能在 Windows 2003 企业版或数据中心版，以及 Windows 2008 企业版及更改版本的操作系统中才能使用。用户可以借助服务器群集功能将多台服务器连接在一起，从而为在该群集中运行的数据和程序提供高可用性和易管理性。

6.7.2　服务器群集配置要求

SQL Server 故障转移群集是建立在 Windows 群集的基础上，在安装 SQL Server 故障转移群集之前必须要确保已经安装并配置好了 Windows 故障转移群集。在服务器上安装 Windows 故障转移群集必须符合下列要求：

1. 软件要求

要配置服务器群集必须具备以下软件条件：

- 群集中的所有计算机均安装了 Microsoft Windows Server 2008 Enterprise Edition 或 Windows Server 2008 Datacenter Edition。
- 一个名称解析法，例如，域名系统（Domain Name System，DNS）、DNS 动态更新协议、Windows Internet 名称服务（Windows Internet Name Service，WINS）、HOSTS 等。
- 一个现有的域模型。
- 所有的节点必须是同一个域的成员。
- 一个域级账户，必须是每个节点上的本地管理员组的成员。建议采用专用账户。

2. 硬件要求

配置服务器群集的硬件要求是：

- 群集硬件必须属于群集服务硬件兼容性列表（Hardware Compatibility List，HCL）。

说明： 要查找最新的群集服务硬件兼容性列表，请登录位于 http://www.microsoft.com/hcl/ 的 Windows 硬件兼容性列表（Windows Hardware Compatibility List），然后搜索“cluster”（群集）。整个解决方案必须得到 HCL 认证，而不仅仅是个别组件。

- 两个超大存储设备控制器——小型计算机系统接口（Small Computer System Interface,

SCSI）或光纤通道（Fibre Channel）。一个用于在其中一个域控制器上安装操作系统（OS）的本地系统磁盘；另一个面向共享磁盘的独立的外围组件互连（ PCI）存储控制器。

- 群集中的每个节点拥有两个 PCI 网络适配器。
- 将共享存储设备附加到所有计算机的存储电缆。
- 对于所有的节点，一切硬件都必须是可识别的，对应正确的插槽、设备卡、BIOS、固件修订版等。这将使配置变得更加简单，同时消除兼容性问题。

注意：如果正在存储区域网络（SAN）上安装该群集，并计划让多个设备和群集与之共享 SAN，那么该解决方案也必须服从“群级/多群集设备（Cluster/Multi-Cluster Device）”硬件兼容性列表。

3．网络要求

在网络上，对服务器群集中的每一台服务器有如下要求：

- 一个唯一的 NetBIOS 名称。
- 每个节点上的所有网络界面均拥有静态 IP 地址。
- 接入一个域控制器。如果群集服务无法验证用于启动服务的用户账户，可能导致群集发生故障。建议在群集所在的相同的局域网（LAN）中配置一个域控制器，以便确保其可用性。
- 每个节点至少必须拥有两个网络适配器，一个用于连接客户端公用网络，另一个用于连接节点对节点的专用群集网络。HCL 认证要求要有一个专用网络适配器。
- 所有节点都必须拥有两个面向公用和专用通信的物理独立的局域网（LAN）或虚拟局域网（LAN）。
- 如果使用容错网卡或网络适配器组合，确认使用最新的固件和驱动程序，并向网络适配器制造商核实群集兼容性。

注意：服务器群集（Server Clustering）不支持使用由动态主机配置协议（Dynamic Host Configuration Protocol，DHCP）服务器分配的地址。

4．共享磁盘要求

服务器群集中必须要有共享磁盘，同时对共享磁盘有以下要求：

- 一个经 HCL 认可的连接到所有计算机的外部磁盘存储单元，其将用做群集共享磁盘。建议采用某种类型的硬件独立磁盘冗余阵列（RAID）。
- 所有共享磁盘，包括仲裁磁盘，必须在物理上附加到一个共享总线。
- 共享磁盘必须位于系统驱动器所用的控制器以外的另一个控制器上。
- 建议在 RAID 配置中创建多个硬件级别的逻辑驱动器，而不是使用一个单一的逻辑磁盘，然后将其分成多个操作系统级别的分区。这不同于独立服务器通常所采用的配置。但是，它可以在群集中拥有多个磁盘资源，并跨节点执行“活动/活动（Active/Active）”配置和手动负载平衡。
- 最小 50 兆字节（MB）的专用磁盘用做仲裁设备。为了得到最佳的 NTFS 文件系

统性能，建议采用最小 500 MB 的磁盘分区。

- 确认可以从所有的节点看到附加到共享总线的磁盘，可以在主适配器安装中进行查看。可以参考制造商的文档，了解适配器指定的指导说明。
- 必须根据制造商的指导说明，对 SCSI 设备分配唯一的 SCSI 标识号，并正确地将其连接。
- 所有共享磁盘必须配置为基本磁盘。
- 群集共享磁盘不支持软件容错。
- 在运行 64 位版本的 Windows Server 2008 的系统上，所有共享磁盘必须配置为主引导记录（MBR）。
- 群集磁盘上的所有分区必须格式化为 NTFS。
- 建议所有磁盘均采用硬件容错 RAID 配置。
- 建议最少采用两个逻辑共享驱动器。

6.7.3　创建 Windows 故障转移群集

在确认了各方面满足服务器群集配置要求后便可创建 SQL Server 故障转移群集。创建数据库故障转移群集主要经过以下几步操作。

（1）创建活动目录（也就是域）。

（2）创建 Windows 故障转移群集。

（3）将 SQL Server 2012 安装盘放入光驱，启用 SQL Server 2012 安装向导，单击“安装”选项下的“新的 SQL Server 故障转移群集安装”选项，如图 6.29 所示。



图 6.29　安装 SQL Server 故障转移群集

（4）根据安装向导选择群集组、进行群集节点配置、群集服务域组配置等，最终完成

SQL Server 2012 故障转移群集的安装。

6.8　小　　结

本章主要从提高系统可用性出发，讲解了在数据文件安全与灾难恢复上 SQL Server 2012 对应的解决方案。

数据文件的安装就在于数据的备份，从冷备、温备和热备 3 个方面 SQL Server 2012 都提供了对应的处理技术。

本章主要侧重于数据库的管理和配置上，其中的数据库镜像、日志传送和数据库群集都是配置比较复杂的，主要是数据库管理人员（DBA）在使用，而且对于开发人员来说这些技术都是透明的。所以读者若是一个 SQL Server 初学者则可以跳过这几个章节，对于普通数据库开发人员来说，只需要了解数据库的备份与恢复、分离与附加即可。

第 7 章　复　　制

SQL Server 复制是一组技术，它将数据和数据库对象从一个数据库复制和分发到另一个数据库，然后在数据库间进行同步，以维持一致性。复制主要用于解决分布式数据问题。本章主要从复制的概念、工作机制到配置和管理等多方面来讲解 SQL Server 复制。

7.1　使用 bcp 导入导出数据

大容量复制程序（bulk copy program，bcp）主要用于导入导出大容量数据。bcp 具有较强的用途，是一个非常紧凑而快捷的数据库工具。

7.1.1　bcp 实现大容量复制

bcp 实用工具是运行在命令行模式下的程序，可以在 SQL Server 实例和用户指定格式的数据文件间大容量复制数据。使用 bcp 实用工具可以将大量新行导入 SQL Server 表，或将表数据导入到数据文件中。除了与 queryout 选项一起使用外，使用 bcp 工具不需要了解 T-SQL 的知识。若要将数据从文件导入到表中，必须使用为该表创建的格式文件，或者必须了解表的结构以及对于该表中的列有效的数据类型。bcp 使用的语法如代码 7.1 所示。

代码 7.1　bcp 语法

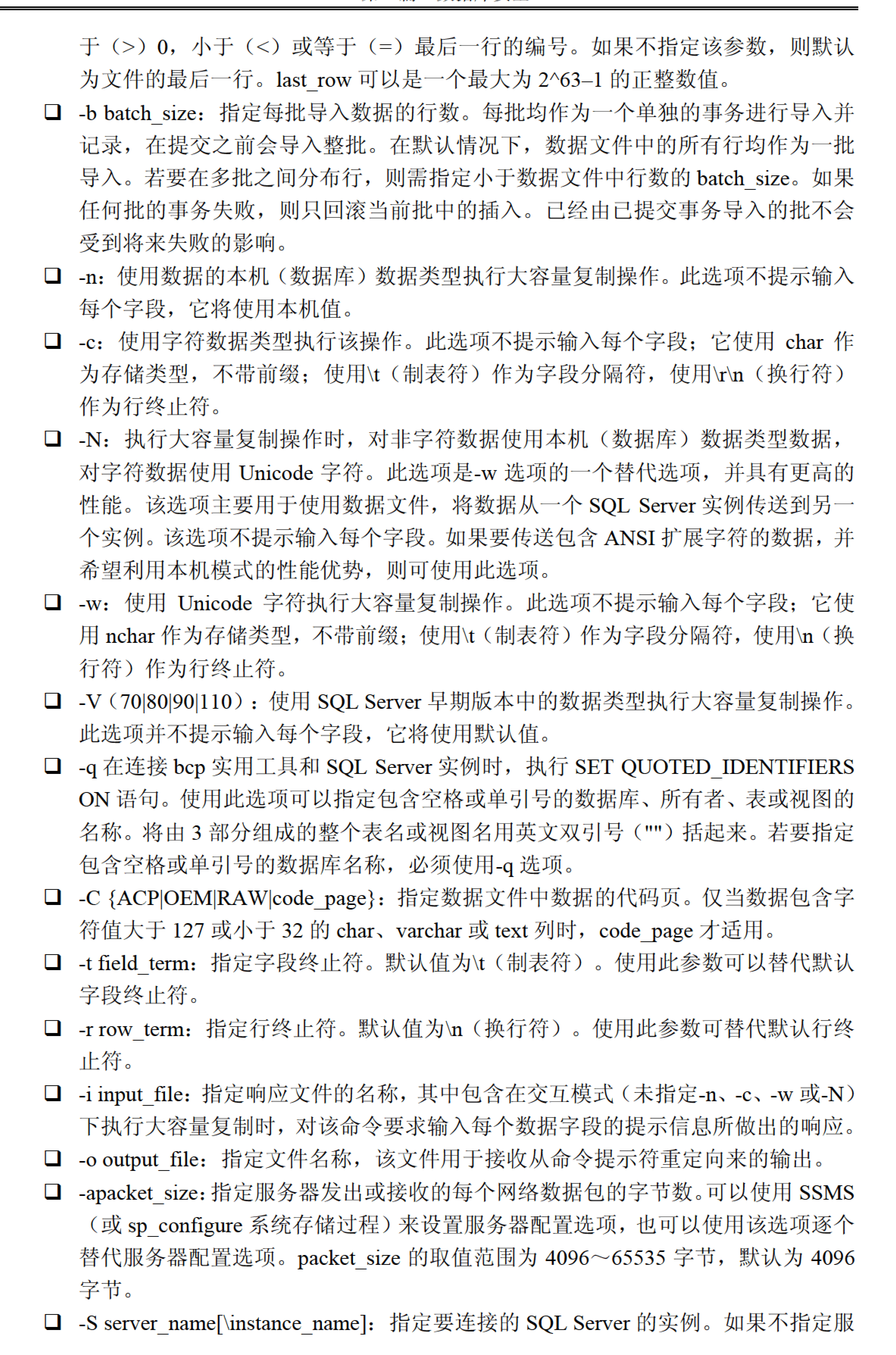
```
bcp {[[database_name.][owner].]{table_name|view_name}|"query"}
{in|out|queryout|format}data_file
[-mmax_errors][-fformat_file][-x][-eerr_file]
[-Ffirst_row][-Llast_row][-bbatch_size]
[-n][-c][-N][-w][-V (70|80|90) ]
[-q][-C{ACP|OEM|RAW|code page}][-tfield term]
[-rrow term][-iinput file][-ooutput file][-apacket size]
[-Sserver name[\instance name]][-Ulogin id][-Ppassword]
[-T][-v][-R][-k][-E][-h"hint[,...n]"]
```

其中各参数的含义如下所述。

- database_name：指定的表或视图所在数据库的名称。如果不指定，则使用用户的默认数据库。
- owner：表或视图所有者的名称。如果执行该操作的用户拥有指定的表或视图，则 owner 是可选的。如果未指定 owner，并且执行该操作的用户不是指定的表或视图的所有者，则 SQL Server 将返回错误消息，而且该操作将取消。
- table_name：将数据导入 SQL Server（in）时的目标表名称，以及将数据从 SQL Server

（out）导出时的源表名称。

- view_name：将数据复制到 SQL Server（in）时的目标视图名称，以及复制 SQL Server（out）中的数据时的源视图名称。只有其中所有列都引用同一个表的视图才能用做目标视图。
- "query"：一个返回结果集的 Transact-SQL 查询。如果该查询返回多个结果集（如包含 COMPUTE 子句的 SELECT 语句），则只将第一个结果集复制到数据文件，而忽略其他结果集。可以将查询放在英文双引号中，将查询中嵌入的任何内容放在英文单引号中。在查询大容量复制数据时，还必须指定 queryout。
- in：指定大容量复制的方向从文件复制到数据库表或视图。
- out：指定大容量复制的方向从数据库表或视图复制到文件。如果指定了现有文件，则该文件将被覆盖。提取数据时要注意，bcp 实用工具将空字符串表示为 null，而将 null 字符串表示为空字符串。
- queryout：指定大容量复制的方向从查询中复制，仅当从查询大容量复制数据时才必须指定此选项。
- format：指定大容量复制的方向根据指定的选项（-n、-c、-w 或-N），以及表或视图的分隔符创建格式化文件。大容量复制数据时，bcp 命令可以引用一个格式文件，从而避免以交互方式重复输入格式信息。format 选项要求指定-f 选项；创建一个 XML 格式文件时还需要指定-x 选项。
- data_file：数据文件的完整路径。将数据大容量导入 SQL Server 时，数据文件将包含要复制到指定表或视图的数据。从 SQL Server 中大容量导出复制数据时，数据文件将包含从表或视图复制的数据。路径可以有 1～255 个字符。数据文件最多可包含 2147483647 行。
- -m max_errors：指定取消 bcp 操作之前可能出现的语法错误的最大数目。语法错误是指将数据转换为目标数据类型时的错误。max_errors 总数不包括只能在服务器中检测到的错误，如违反约束。无法由 bcp 实用工具复制的行将被忽略，并计为一个错误。如果不指定此选项，则默认为 10。
- -f format_file：指定一个格式文件的完整路径。该选项的含义取决于使用它的环境。如果-f 与 format 选项一起使用，则将为指定的表或视图创建指定的 format_file。若要创建 XML 格式文件，请同时指定-x 选项。如果与 in 或 out 一起使用，则应为-f 指定一个现有的格式文件。
- -x 与 format 和-f format_file 选项一起使用，可以生成基于 XML 的格式化文件，而不是默认的非 XML 格式化文件。在导入或导出数据时，-x 不起作用。如果不与 format 和-f format_file 一起使用，则将生成错误。
- -e err_file：指定错误文件的完整路径，此文件用于存储 bcp 无法从文件传输到数据库的所有行。bcp 命令产生的错误消息将被发送到用户的工作站。如果不使用此选项，则不会创建错误文件。
- -F first_row：指定要从表中导出或从数据文件导入的第一行的编号。此参数应大于（>）0，小于（<）或等于（=）总行数。如果不指定此参数，则默认为文件的第一行。first_row 可以是一个最大为 2^63–1 的正整数值。
- -L last_row：指定要从表中导出或从数据文件导入的最后一行的编号。此参数应大



于（>）0，小于（<）或等于（=）最后一行的编号。如果不指定该参数，则默认为文件的最后一行。last_row 可以是一个最大为 2^63–1 的正整数值。

- -b batch_size：指定每批导入数据的行数。每批均作为一个单独的事务进行导入并记录，在提交之前会导入整批。在默认情况下，数据文件中的所有行均作为一批导入。若要在多批之间分布行，则需指定小于数据文件中行数的 batch_size。如果任何批的事务失败，则只回滚当前批中的插入。已经由已提交事务导入的批不会受到将来失败的影响。
- -n：使用数据的本机（数据库）数据类型执行大容量复制操作。此选项不提示输入每个字段，它将使用本机值。
- -c：使用字符数据类型执行该操作。此选项不提示输入每个字段；它使用 char 作为存储类型，不带前缀；使用\t（制表符）作为字段分隔符，使用\r\n（换行符）作为行终止符。
- -N：执行大容量复制操作时，对非字符数据使用本机（数据库）数据类型数据，对字符数据使用 Unicode 字符。此选项是-w 选项的一个替代选项，并具有更高的性能。该选项主要用于使用数据文件，将数据从一个 SQL Server 实例传送到另一个实例。该选项不提示输入每个字段。如果要传送包含 ANSI 扩展字符的数据，并希望利用本机模式的性能优势，则可使用此选项。
- -w：使用 Unicode 字符执行大容量复制操作。此选项不提示输入每个字段；它使用 nchar 作为存储类型，不带前缀；使用\t（制表符）作为字段分隔符，使用\n（换行符）作为行终止符。
- -V（70|80|90|110）：使用 SQL Server 早期版本中的数据类型执行大容量复制操作。此选项并不提示输入每个字段，它将使用默认值。
- -q 在连接 bcp 实用工具和 SQL Server 实例时，执行 SET QUOTED_IDENTIFIERS ON 语句。使用此选项可以指定包含空格或单引号的数据库、所有者、表或视图的名称。将由 3 部分组成的整个表名或视图名用英文双引号（""）括起来。若要指定包含空格或单引号的数据库名称，必须使用-q 选项。
- -C {ACP|OEM|RAW|code_page}：指定数据文件中数据的代码页。仅当数据包含字符值大于 127 或小于 32 的 char、varchar 或 text 列时，code_page 才适用。
- -t field_term：指定字段终止符。默认值为\t（制表符）。使用此参数可以替代默认字段终止符。
- -r row_term：指定行终止符。默认值为\n（换行符）。使用此参数可替代默认行终止符。
- -i input_file：指定响应文件的名称，其中包含在交互模式（未指定-n、-c、-w 或-N）下执行大容量复制时，对该命令要求输入每个数据字段的提示信息所做出的响应。
- -o output_file：指定文件名称，该文件用于接收从命令提示符重定向来的输出。
- -apacket_size：指定服务器发出或接收的每个网络数据包的字节数。可以使用 SSMS（或 sp_configure 系统存储过程）来设置服务器配置选项，也可以使用该选项逐个替代服务器配置选项。packet_size 的取值范围为 4096～65535 字节，默认为 4096 字节。
- -S server_name[\instance_name]：指定要连接的 SQL Server 的实例。如果不指定服

务器，则 bcp 实用工具将连接到本地计算机上的默认 SQL Server 实例。如果从网络或本地命名实例上的远程计算机运行 bcp 命令，则必须使用此选项。若要连接到服务器的 SQL Server 默认实例，仅需指定 server_name 即可。若要连接到 SQL Server 的命名实例，则指定 server_name\instance_name。

- -U login_id：指定用于连接 SQL Server 的登录 ID。

说明： 如果 bcp 实用工具使用集成安全性的可信连接与 SQL Server 进行连接，则使用-T 选项（可信连接），而不要使用 username 和 password 组合。

- -P password 指定登录 ID 的密码。如果不使用此选项，bcp 命令将提示输入密码。如果在命令提示符的末尾使用此选项，但不提供密码，则 bcp 将使用默认密码（NULL）。

技巧： 若要屏蔽密码，请不要同时使用-U 和-P 选项，而应在指定 bcp 以及-U 选项和其他开关（不指定-P）之后，按 Enter 键，这时命令会提示输入密码。这种方法可以确保密码在输入时被屏蔽。

- -T：指定 bcp 实用工具通过使用集成安全性的可信连接连接到 SQL Server。不需要网络用户的安全凭据、login_id 和 password。如果不指定-T，则需要指定-U 和-P 才能成功登录。
- -v：报告 bcp 实用工具的版本号和版权。
- -R：指定使用客户端计算机区域设置中定义的区域格式，将货币、日期和时间数据大容量复制到 SQL Server 中。在默认情况下，将忽略区域设置。
- -k：指定在操作过程中空列应该保留空值，而不是所插入列的任何默认值。
- -E：指定导入数据文件中的标识值用于标识列。如果未指定-E，则将忽略所导入数据文件中此列的标识值，而且 SQL Server 将根据创建表期间指定的种子值和增量值自动分配唯一值。假如数据文件不包含表或视图中的标识列的值，则可在格式文件中指定，在导入数据时忽略表或视图中的标识列；SQL Server 将自动为该列分配唯一值。
- -h "hint[,...n]"：指定向表或视图中大容量导入数据时所用的提示。

在 SQL Server 2012 中，bcp 实用工具仅支持与 SQL Server 7.0、SQL Server 2000、SQL Server 2005 和 SQL Server 2008 兼容的本机数据文件。SQL Server 2012 不支持 SQL Server 7.0 版本之前的数据文件。

7.1.2　bcp 导出

大容量数据操作分为数据的导入和导出。bcp 允许从表、视图和查询中导出数据。bcp 导出时必须指定目的文件名，如果文件已经存在，则会被覆盖。在导出的时候不允许跳过列。另外要运行导出则必须拥有源表的 SELECT 权限。

下面以 AdventureWorks2012 数据库为例，需要将表 Person.AddressType 中的所有数据导出为数据文件 AddressType.dat，若使用 Windows 身份认证进行受信任的连接，那么将该表的内容导出的 bcp 命令如代码 7.2 所示。

代码 7.2　bcp 导出数据

```
bcp AdventureWorks2012.Person.AddressType out AddressType.dat -T -c
```

这里只指定了输出文件名为 AddressType.dat，并没有指定该文件的路径，这种情况下，系统将在当前运行的路径创建文件。若不希望在当前路径创建文件可以指定输出文件的完整路径。运行 bcp 导出的结果如图 7.1 所示。

若要连接的是远程数据库，需要使用用户名密码并且不希望密码在命令行中被显示出来，那么就需要使用 bcp 的提示信息一步步地完成数据导出的操作。例如要连接远程数据库 10.101.10.66，同样是导出 AdventureWorks2012 数据库中的 Person.AddressType 表，那么导出的 bcp 命令如代码 7.3 所示。

代码 7.3　bcp 导出远程数据库数据

```
bcp  AdventureWorks2012.Person.AddressType  out  AddressType.dat  -c  -S
10.101.10.66 -U sa
```

运行后系统提示输入密码，此时密码的输入是不可见的，输入密码后回车即可开始 bcp 导出，如图 7.2 所示。

```
D:\Program Files\Microsoft SQL Server\110\Tools\Binn>bcp AdventureWorks2012.Pers
on.AddressType out AddressType.dat -T -c

开始复制...

已复制 6 行。
网络数据包大小(字节): 4096
总时钟时间(毫秒)     : 109      平均值: (每秒 55.05 行。)

D:\Program Files\Microsoft SQL Server\110\Tools\Binn>_
```

图 7.1　运行 bcp 导出数据

```
D:\Program Files\Microsoft SQL Server\110\Tools\Binn>bcp AdventureWorks2012.Pers
on.AddressType out AddressType.dat -c -S 10.101.10.66 -U sa
密码:
开始复制...

已复制 6 行。
网络数据包大小(字节): 4096
总时钟时间(毫秒)     : 109      平均值: (每秒 55.05 行。)

D:\Program Files\Microsoft SQL Server\110\Tools\Binn>_
```

图 7.2　bcp 导出远程服务器数据

7.1.3　格式化文件

在数据导出过程中，需要将数据库表中各字段的格式信息导出。用于存储与特定表相关的数据文件中，各字段格式信息的文件称为格式化文件。格式化文件中的数据提供了大容量导出或大容量导入数据所需的全部格式信息。格式化文件只需用户极少的编辑，甚至不需要编辑，即可写入符合其他数据格式的数据文件，或从其他软件读取数据文件。

1．格式化文件简介

在 SQL Server 2000 和更早期的版本中，大容量导出和导入使用的是单一类型的格式化文件，在 SQL Server 2012 中仍然支持这种文件。在 SQL Server 2005 中又添加了对 XML 格式化文件的支持，做备选之用。为了便于区分，原来类型的格式化文件称为“非 XML 格式化文件”。

每个格式化文件中都包含对数据文件中每个字段的说明。而且 XML 格式化文件还包

含对相应表列的说明。一般情况下，XML 与非 XML 格式化文件可以互换，但建议为新的格式化文件使用 XML 语法，因为与非 XML 格式化文件相比，格式化文件具有多项优点。

XML 格式化文件具有以下特点：

- 自描述且易于读取、创建和扩展。直接阅读 XML 格式化文件便可了解其中的意思。
- 包含目标列的数据类型。
- 允许从数据文件加载包含单一大型对象（LOB）数据类型的字段。

对于 bcp 或 BULK INSERT，由于格式化文件是可选的，在简单的情况下很少使用。但是，对于复杂的大容量导入情况，通常都会需要格式化文件，例如，在将数据从数据文件导入到表中时。除此之外对于以下情况可能需要格式化文件：

- 用户对目标表的某些列没有 INSERT 权限。
- 具有不同架构的多个表使用同一数据文件作为数据源。
- 数据文件中的列与目标表中的列顺序不同。
- 数据文件中的数据元素包含不同的终止字符或前缀长度。

如果出现以下情况，则必须使用格式化文件：

- 数据文件中的字段数不同于目标表中的列数。
- 目标表中至少包含一个定义了默认值或允许为 NULL 的列。
- 用户不具有对目标表的一个或多个列的 SELECT/INSERT 权限。
- 具有不同架构的两个或多个表使用同一个数据文件。
- 数据文件和表的列顺序不同。
- 数据文件列的终止字符或前缀长度不同。

在 7.1.2 节中我们已经将数据导出到 AddressType.dat 文件中，用记事本或其他文本编辑器打开该文件，可以看到导出的内容如代码 7.4 所示。

代码 7.4　导出的内容

```
1    Billing B84F78B1-4EFE-4A0E-8CB7-70E9F112F886   1998-06-01
     00:00:00.000
2    Home 41BC2FF6-F0FC-475F-8EB9-CEC0805AA0F2  1998-06-01  00:00:00.000
3    Main Office 8EEEC28C-07A2-4FB9-AD0A-42D4A0BBC575   1998-06-01
     00:00:00.000
4    Primary 24CB3088-4345-47C4-86C5-17B535133D1E   1998-06-01
     00:00:00.000
5    Shipping B29DA3F8-19A3-47DA-9DAA-15C84F4A83A5  1998-06-01
     00:00:00.000
6    Archive A67F238A-5BA2-444B-966C-0467ED9C427F   1998-06-01
     00:00:00.000
```

从该内容中无法看出每个字段的数据类型定义和数据长度，以及其他限制。

2．非XML格式化文件

若以非 XML 格式化文件的形式获得源数据的格式，则对应的命令如代码 7.5 所示。

代码 7.5　导出非 XML 格式化文件

```
bcp AdventureWorks.Person.AddressType format nul -f AddressType.fmt -n -T
以下是输出文件的内容
10.0
4
1    SQLINT       0     4     ""   1    AddressTypeID        ""
2    SQLNCHAR     2     100   ""   2    Name SQL_Latin1_General_CP1_CI_AS
```

```
3    SQLUNIQUEID  1     16     ""    3    rowguid            ""
4    SQLDATETIME  0     8      ""    4    ModifiedDate       ""
```

整个格式文件的字段如图 7.3 所示。

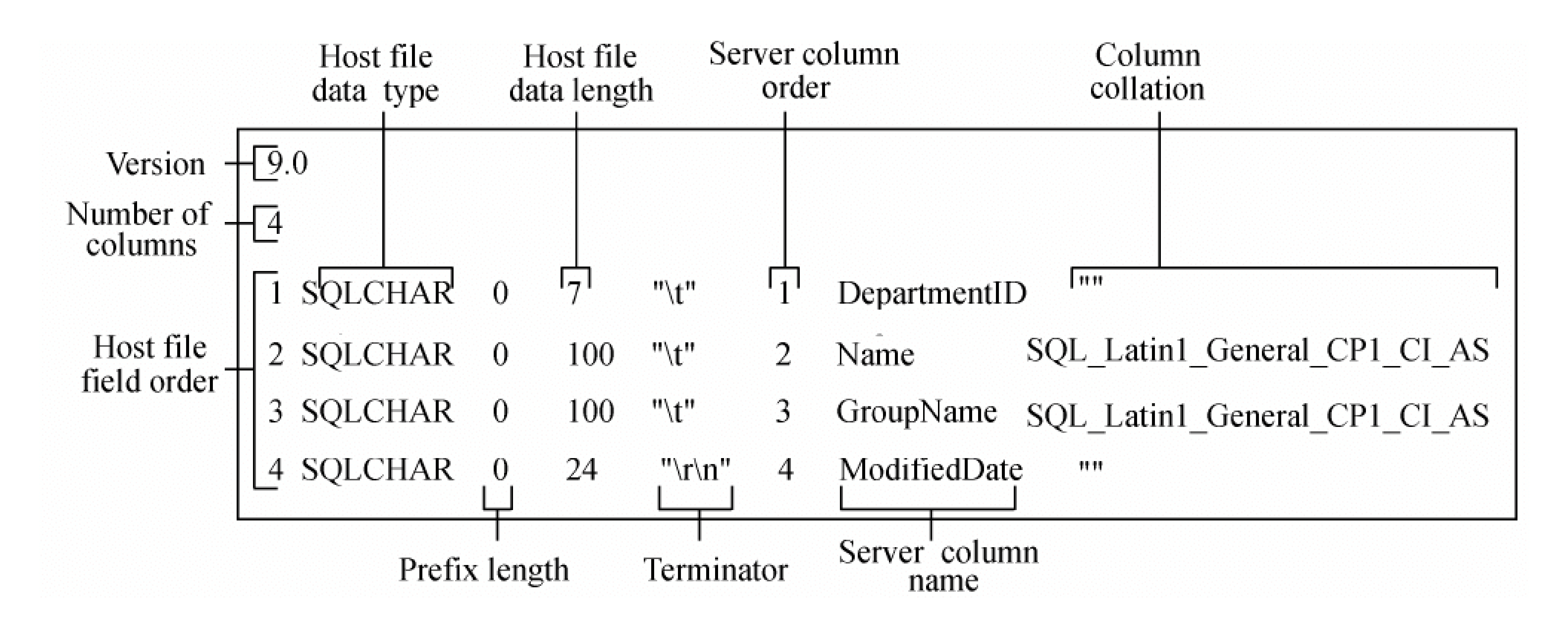


图 7.3　非 XML 格式文件的字段说明

非 XML 格式化文件中第 1～10 行指出了当前导出的 SQL Server 版本为 SQL Server 2008，在 bcp 中各版本号对应是：

- 7.0 = SQL Server 7.0 版；
- 8.0 = SQL Server 2000；
- 9.0 = SQL Server 2005；
- 10.0 = SQL Server 2008。

第 2 行的 4 表示总共导出了 4 列，接下来的 4 行是说明每列的数据类型前缀长度等。具体说明如表 7.1 所示。

表 7.1　格式化文件字段说明

格式化文件字段	说　　明
宿主文件字段顺序	用以表示数据文件中每个字段位置的数字。行中的第一个字段为1，依次类推
宿主文件数据类型	表示存储在数据文件给定字段中的数据类型。对于ASCII数据文件，使用SQLCHAR；对于本机格式数据文件，使用默认的数据类型
前缀长度	字段长度前缀字符的数目。有效前缀长度是0、1、2、4和8。若要避免指定长度前缀，将其设置为0。如果字段包含NULL数据值，则必须指定长度前缀
宿主文件数据长度	数据文件的特定字段中所存储的数据类型的最大长度（按字节计）。如果正在为带分隔符的文本文件创建非XML格式化文件，则可以将每个数据字段的宿主文件数据长度指定为0。当带分隔符的文本文件的前缀长度为 0 并导入终止符时，可忽略字段长度值，因为字段所使用的存储空间等于数据加上终止符的长度
终止符	用来分隔数据文件中各字段的分隔符。常用的终止符为逗号（,）、制表符（\t）和行结束符（\r\n）
服务器列顺序	列在SQL Server表中显示的顺序。例如，如果数据文件的第4个字段映射到SQL Server表中的第4列，则第4个字段的服务器列顺序为6。若要阻止表中的某个列接收数据文件中的任何数据，则可以将服务器列顺序值设置为0
服务器列名	从SQL Server表中复制的列名。无需使用字段的实际名称，但格式化文件中的字段不得为空
列排序规则	排序规则用于在数据文件中存储字符和Unicode数据

3．XML格式化文件

若要以 XML 格式化文件则需要使用-x 参数，如代码 7.6 为导出 XML 格式化文件的命令和导出的 XML 结果。

代码 7.6 导出 XML 格式化文件命令和结果

```
bcp AdventureWorks.Person.AddressType format nul -f AddressType.xml -x -n
-T
以下是导出的文件的内容
<?xml version="1.0"?>
<BCPFORMAT                          xmlns="http://schemas.microsoft.com/SQL
Server/2004/bulkload/format"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <RECORD>
  <FIELD ID="1" xsi:type="NativeFixed" LENGTH="4"/>
  <FIELD ID="2" xsi:type="NCharPrefix" PREFIX LENGTH="2" MAX LENGTH="100"
COLLATION="SQL Latin1 General CP1 CI AS"/>
  <FIELD ID="3" xsi:type="NativePrefix" PREFIX_LENGTH="1"/>
  <FIELD ID="4" xsi:type="NativeFixed" LENGTH="8"/>
 </RECORD>
 <ROW>
  <COLUMN SOURCE="1" NAME="AddressTypeID" xsi:type="SQLINT"/>
  <COLUMN SOURCE="2" NAME="Name" xsi:type="SQLNVARCHAR"/>
  <COLUMN SOURCE="3" NAME="rowguid" xsi:type="SQLUNIQUEID"/>
  <COLUMN SOURCE="4" NAME="ModifiedDate" xsi:type="SQLDATETIME"/>
 </ROW>
</BCPFORMAT>
```

XML 格式化文件具有两个主要组件：<RECORD>和<ROW>。

<RECORD>说明数据文件中存储的数据。每一<RECORD>元素包含一个或多个<FIELD>元素。这些元素与数据文件中的字段相对应。基本语法如下：

```
<RECORD>
   <FIELD .../> [ ...n ]
</RECORD>
```

每个<FIELD>元素说明特定数据字段的内容。一个字段只能映射到表中的一列，并不是所有字段都需要映射到列。

数据文件中字段的长度可以是固定或可变的，也可以由字符结尾。“字段值”可以表示为字符（使用单字节表示形式）、宽字符（使用 Unicode 双字节表示形式）、本机数据库格式或文件名。如果字段值为文件名，则文件名指向包含目标表中 BLOB 列值的文件。

<ROW>说明在将数据从文件导入 SQL Server 表中时，如何构造数据文件中的数据行。

<ROW>元素包含一组<COLUMN>元素。这些元素与表列相对应。基本语法如下：

```
<ROW>
   <COLUMN .../> [ ...n ]
</ROW>
```

一列只能与数据文件中的一个字段相映射。

7.1.4　bcp 导入

前面已经了解了 bcp 导出和格式化文件等基础知识，本节主要实现使用 bcp 将大量数据导入到 SQL Server 表和视图中。要导入数据到 SQL Server，必须要有对目的表或视图的 INSERT 和 SELECT 权限。

bcp 操作忽略了规则，除非指定了 FIRE_TRIGGERS 或 CHECK_CONSTRAINTS 的指示，否则将忽略所有触发器和约束。bcp 操作时强制执行唯一约束、索引和主外键约束，除非指定了-K 选项；否则执行默认约束。

以前面导出的数据 AddressType.dat 为例，现在需要将该文件中的数据重新导入到数据库中，只是此时要将数据导入到 TestDB1 数据库中。首先需要在 TestDB1 中建立 AddressType 表，在此之前若没有创建 TestDB1 数据库则先创建数据库，建立表可以使用 SELECAT * INTO 命令。具体脚本如代码 7.7 所示。

代码 7.7　创建表

```
USE TestDB1;
GO
SELECT * INTO dbo.AddressType
FROM AdventureWorks2012.Person.AddressType
WHERE 1=2
```

现在表 dbo.AddressType 已经创建，下一步就是使用 bcp 命令将数据文件中的数据导入到该表中。导入使用 bcp 的 in 参数，具体命令如代码 7.8 所示。

代码 7.8　bcp 导入数据

```
bcp TestDB1.dbo.AddressType in AddressType.dat -T -c
```

导入完成后系统会显示导入的行数、耗时等，如图 7.4 所示。

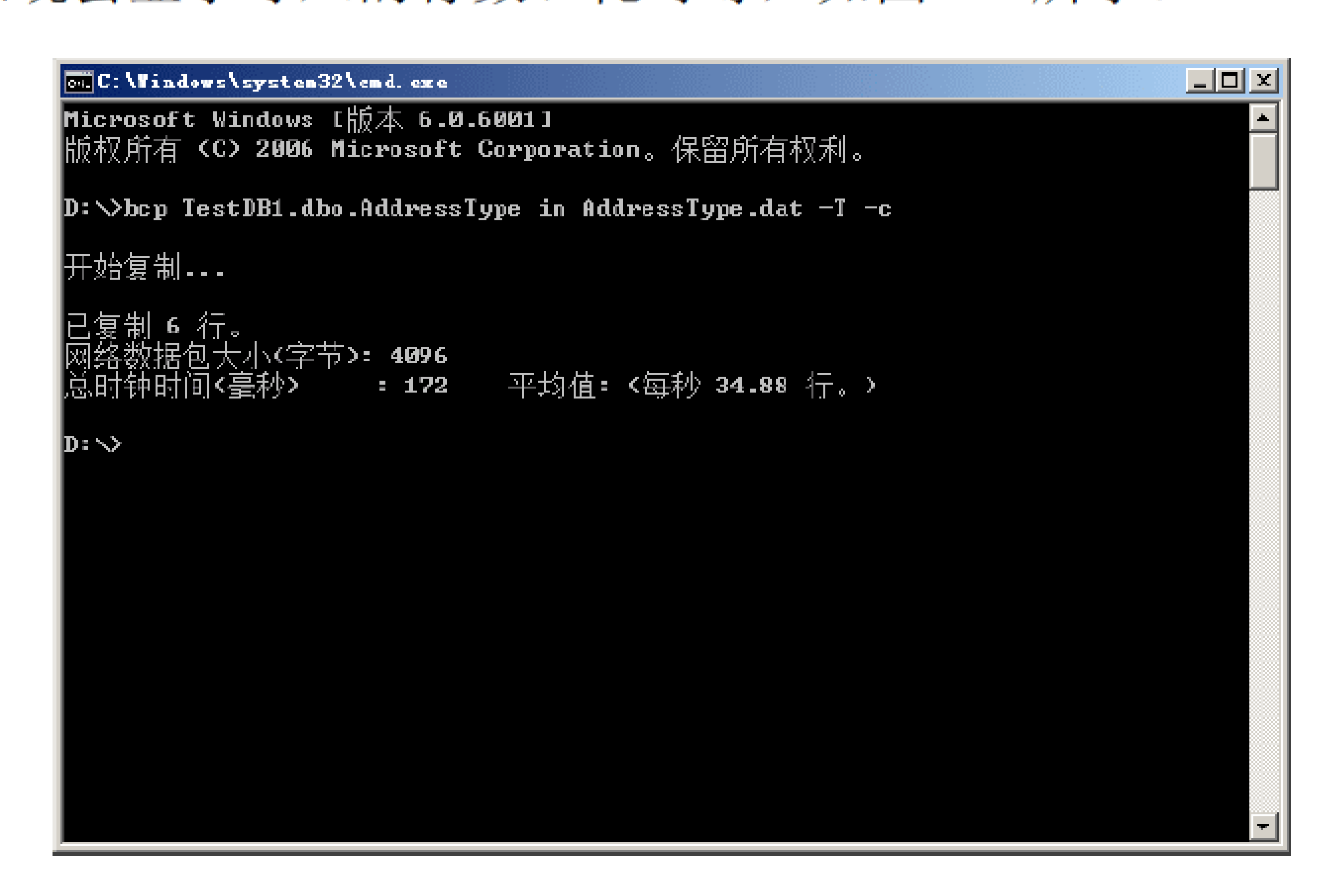


图 7.4　bcp 导入的结果

若要复制特定列，可以使用 queryout 选项。以下示例将 Person.AddressType 表中的 Name 列复制到数据文件中。这里都使用 Windows 身份认证并且使用可信任的连接。导出 Name 列只需要在命令行下输入：

```
bcp "select Name from AdventureWorks.Person.AddressType" queryout
ATName.dat -T -c
```

复制出的数据只包含了 Name 列的内容，要将该数据导入，仍然使用 bcp 命令带 in 参数即可。

在向 SQL Server 的实例导入数据时，若要使用以前创建的格式文件，可同时使用-f 开关和 in 选项。例如，通过使用先前创建的格式化文件（AddressType.xml），将数据文件 AddressType.dat 的内容大容量复制到 TestDB1 数据库的 AddressType2 表中，则只需要在命令行下输入：

```
bcp TestDB1.dbo.AddressType2 in AddressType.dat -T -f AddressType.xml
```

bcp 可以运行在快模式（不记录日志）和慢模式（记录日志）下。快模式的优点是提供了最好的性能，慢模式的优点是提供了最大的可恢复性。因为慢模式是记录日志的，所以可以在导入后运行一个快速事务日志并能在发生错误时恢复数据库。

7.1.5　使用 BULK INSERT 命令

要使用 SQL Server 的 BULK INSERT 命令，则必须是 sysadmin 或 bulkadminserver 角色的成员。BULK INSERT 在本质上与直接在 T-SQL 中可用的 bcp 限制版上的操作方式一样。其语法如代码 7.9 所示。

代码 7.9　BULK INSERT 语法

```
BULK INSERT
   [ database_name . [ schema_name ] . | schema_name . ] [ table_name |
   view_name ]
      FROM 'data_file'
     [ WITH
        (
   [ [ , ] BATCHSIZE = batch_size ]
      [ [ , ] CHECK_CONSTRAINTS ]
   [ [ , ] CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'code_page' } ]
      [ [ , ] DATAFILETYPE =
      { 'char' | 'native'| 'widechar' | 'widenative' } ]
   [ [ , ] FIELDTERMINATOR = 'field_terminator' ]
   [ [ , ] FIRSTROW = first_row ]
   [ [ , ] FIRE_TRIGGERS ]
   [ [ , ] FORMATFILE = 'format_file_path' ]
   [ [ , ] KEEPIDENTITY ]
   [ [ , ] KEEPNULLS ]
   [ [ , ] KILOBYTES_PER_BATCH = kilobytes_per_batch ]
   [ [ , ] LASTROW = last_row ]
   [ [ , ] MAXERRORS = max_errors ]
   [ [ , ] ORDER ( { column [ ASC | DESC ] } [ ,...n ] ) ]
   [ [ , ] ROWS_PER_BATCH = rows_per_batch ]
   [ [ , ] ROWTERMINATOR = 'row_terminator' ]
   [ [ , ] TABLOCK ]
   [ [ , ] ERRORFILE = 'file_name' ]
        )]
```

其中各个参数的含义如下所述。

- database_name：包含指定表或视图的数据库名称。如果未指定，则默认为当前数

据库。

- ❑ schema_name：表或视图架构的名称。如果用户执行大容量导入操作的默认架构为指定表或视图的架构，则 schema_name 是可选的。如果未指定 schema 并且用户执行大容量导入操作的默认架构与指定表或视图的架构不同，则 SQL Server 将返回一条错误消息，同时取消大容量导入操作。
- ❑ table_name：要将数据大容量导入其中的表或视图的名称。只能使用其所有列均引用相同基表的视图。
- ❑ 'data_file'：数据文件的完整路径，该数据文件包含要导入到指定表或视图中的数据。使用 BULKINSERT 可以从磁盘（包括网络、软盘、硬盘等）导入数据。
- ❑ data_file：必须基于运行 SQL Server 服务器指定的有效路径。如果 data_file 为远程文件，则指定通用命名约定（UNC）名称。
- ❑ BATCHSIZE=batch_size：指定批处理中的行数。每个批处理作为一个事务复制到服务器中。如果复制操作失败，则 SQL Server 提交或回滚每个批处理的事务。默认情况下，指定数据文件中的所有数据为一个批处理。
- ❑ CHECK_CONSTRAINTS：指定在大容量导入操作期间，必须检查所有对目标表或视图的约束。若没有 CHECK_CONSTRAINTS 选项，则所有 CHECK 和 FOREIGNKEY 约束都将被忽略，并且在此操作之后表的约束将标记为不可信。

注意：无论什么情况下，始终强制使用 UNIQUE、PRIMARY KEY 和 NOT NULL 约束。

- ❑ CODEPAGE={'ACP'|'OEM'|'RAW'|'code_page'}：指定该数据文件中数据的代码页。仅当数据含有字符值大于 127 或小于 32 的 char、varchar 或 text 列时，CODEPAGE 才适用。
- ❑ DATAFILETYPE={'char'|'native'|'widechar'|'widenative'}：指定 BULKINSERT 使用指定的数据文件类型值执行导入操作。
- ❑ FIELDTERMINATOR='field_terminator'：指定要用于 char 和 widechar 数据文件的字段终止符。默认的字段终止符是\t（制表符）。
- ❑ FIRSTROW=first_row：指定要加载的第一行的行号。默认值是指定数据文件中的第一行。

注意：FIRSTROW 属性不可用于跳过列标题。BULKINSERT 语句不支持跳过标题。跳过行时，SQL Server 数据库引擎只考虑字段终止符，而不会对所跳过行的字段中的数据进行验证。

- ❑ FIRE_TRIGGERS：指定将在大容量导入操作期间执行目标表中定义的所有插入触发器。如果在目标表中为 INSERT 操作定义了触发器，则会对每个完成的批处理触发触发器。如果没有指定 FIRE_TRIGGERS，将不执行任何插入触发器。
- ❑ FORMATFILE='format_file_path'：指定一个格式化文件的完整路径。格式化文件用于说明包含存储响应的数据文件，这些存储响应是使用 bcp 实用工具在相同的表或视图中创建的。

说明：在下列情况下应使用格式化文件：数据文件包含的列多于或少于表或视图包含的列及列的顺序不同，列分隔符发生变化，数据格式有其他更改等。通常，使用 bcp 实用工具创建格式化文件并根据需要用文本编辑器进行修改。

- KEEPIDENTITY：指定导入数据文件中的标识值用于标识列。如果没有指定 KEEPIDENTITY，则此列的标识值可被验证但不能导入，并且 SQL Server 将根据表创建时指定的种子值和增量值自动分配一个唯一的值。如果数据文件不包含该表或视图中标识列的值，则使用一个格式化文件指定在导入数据时表或视图中的标识列被忽略；SQL Server 会自动为此列分配唯一的值。
- KEEPNULLS：指定在大容量导入操作期间空列应保留一个 Null 值，而不插入用于列的任何默认值。
- KILOBYTES_PER_BATCH=kilobytes_per_batch：将每个批处理中数据的近似千字节数（KB）指定为 kilobytes_per_batch。默认情况下，KILOBYTES_PER_BATCH 未知。
- LASTROW=last_row：指定要加载的最后一行的行号。默认值为 0，表示指定数据文件中的最后一行。
- MAXERRORS=max_errors：指定允许在数据中出现的最多语法错误数，超过该数量后将取消大容量导入操作。大容量导入操作未能导入的每一行都将被忽略并且计为一个错误。如果未指定 max_errors，则默认值为 10。

注意：MAX_ERRORS 选项不适用于约束检查，也不适用于转换 money 和 bigint 数据类型。

- ORDER（{column[ASC|DESC]}[,...n]）：指定数据文件中的数据如何排序。如果根据表中的聚集索引（如果有的话）对要导入的数据排序，则可提高大容量导入的性能。如果数据文件按不同于聚集索引键的顺序排序，或者该表没有聚集索引，则忽略 ORDER 子句。提供的列名必须是目标表中有效的列名。在默认情况下，大容量插入操作假设数据文件未排序。对于优化大容量导入，SQL Server 还将验证导入的数据是否已排序。
- n：指示可以指定多个列的占位符。
- ROWS_PER_BATCH=rows_per_batch：指示数据文件中近似的数据行数量。默认情况下，数据文件中所有的数据都作为单一事务发送到服务器，批处理中的行数对于查询优化器是未知的。如果指定了 ROWS_PER_BATCH（其值>0），则服务器将使用该值优化大容量导入操作。为 ROWS_PER_BATCH 指定的值应当与实际行数大致相同。
- ROWTERMINATOR='row_terminator'：指定对于 char 和 widechar 数据文件要使用的行终止符。默认行终止符为\r\n（换行符）。
- TABLOCK：指定为大容量导入操作持续时间获取一个表级锁。如果表没有索引并且指定了 TABLOCK，则该表可以同时由多个客户端加载。默认情况下，锁定行为由表选项 tablelockonbulkload 确定。在大容量导入操作期间持有锁会减少表上的锁争用，从而显著提高操作性能。

- ERRORFILE='file_name'：指定用于收集格式有误且不能转换为 OLEDB 行集的行的文件。这些行将按原样从数据文件复制到此错误文件中。

错误文件是执行命令时创建的。如果文件已经存在则会发生错误。此外，还创建了一个扩展名为.ERROR.txt 的控制文件。此文件引用错误文件中的每一行并提供错误诊断。纠正错误后即可加载数据。

从参数中可以看出，BULK INSERT 与 bcp 命令并没有太大的区别。但是相比 bcp，BULK INSERT 有以下优点：

- 可以使用事务，如 BEGIN TRAN 和相关事务语句进行事务操作。
- 对 SQL Server 来说是运行在 T-SQL 脚本批处理中，可以获得更好的性能。
- 没有 bcp 的语法那么复杂。

下面仍然以前面导出的数据文件 AddressType.dat 为例，先将该数据文件中的数据通过 BULK INSERT 的方式导入到 SQL Server 数据库中。具体脚本如代码 7.10 所示。

代码 7.10　BULK INSERT 导入数据

```
BULK INSERT TestDB1.dbo.AddressType2
FROM 'D:\AddressType.dat'
WITH (
        FIELDTERMINATOR ='\t',  --列与列之间用 Tab 分隔
        ROWTERMINATOR ='\n'     --行与行之间是用换行分隔
    )
```

7.1.6　使用 OPENROWSET()函数

OPENROWSET()函数通过 OLE DB 访问接口连接到远程数据源并从该数据源访问远程数据。OPENROWSET()函数中还引入了大容量行集提供程序，大容量行集提供程序是 OPENROWSET()函数的一种特殊形式，适用于远程数据存放在数据文件中的情况。

OPENROWSET 的语法如代码 7.11 所示。

代码 7.11　OPENROWSET()函数的语法

```
OPENROWSET
( { 'provider_name' , { 'datasource' ; 'user_id' ; 'password'
   | 'provider_string' }
     , {   [ catalog. ] [ schema. ] object
       | 'query'
     }
   | BULK 'data_file' ,
       { FORMATFILE = 'format_file_path' [ <bulk_options> ]
       | SINGLE_BLOB | SINGLE_CLOB | SINGLE_NCLOB }
} )
<bulk_options> ::=
   [ , CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'code_page' } ]
   [ , ERRORFILE = 'file_name' ]
   [ , FIRSTROW = first_row ]
   [ , LASTROW = last_row ]
   [ , MAXERRORS = maximum_errors ]
   [ , ROWS_PER_BATCH = rows_per_batch ]
   [ , ORDER ( { column [ ASC | DESC ] } [ ,...n ] ) [ UNIQUE ]
```

其中各参数的含义如下所述。

- 'provider_name'：表示在注册表中指定的 OLEDB 访问接口的友好名称（或 PROGID）。provider_name 没有默认值。
- 'datasource'：对应于特定 OLEDB 数据源的字符串常量。datasource 是要传递给访问接口的 IDBProperties 接口的 DBPROP_INIT_DATASOURCE 属性，该属性用于初始化访问接口。通常，该字符串包含数据库文件的名称、数据库服务器的名称，或者访问接口能理解的用于定位数据库的名称。
- 'user_id'：是传递给指定 OLEDB 访问接口的用户名。user_id 为连接指定安全上下文，并作为 DBPROP_AUTH_USERID 属性传入以初始化访问接口。user_id 不能是 Windows 登录名称。
- 'password'：是传递给 OLEDB 访问接口的用户密码。在初始化访问接口时，password 作为 DBPROP_AUTH_PASSWORD 属性传入。password 不能是 Windows 密码。
- 'provider_string'：访问接口特定的连接字符串，作为 DBPROP_INIT_PROVIDERSTRING 属性传入以初始化 OLEDB 访问接口。通常，provider_string 封装初始化访问接口所需的所有连接信息。
- catalog：指定对象所在的目录或数据库的名称。
- schema：架构的名称或指定对象的对象所有者名称。
- object：对象名，它唯一地标识出将要操作的对象。
- 'query'：是发送到访问接口并由访问接口执行的字符串。SQL Server 的本地实例不处理该查询，但处理由访问接口返回的查询结果（传递查询）。有些访问接口并不通过表名而是通过命令语言提供其表格格式数据，将传递查询用于这些访问接口是非常有用的。只要查询访问接口支持 OLE DB Command 对象及其强制接口，那么在远程服务器上就支持传递查询。
- BULK 使用 OPENROWSET()的 BULK 行集访问接口读取文件中的数据。在 SQL Server 中，OPENROWSET()无需将数据文件中的数据加载到目标表便可读取这些数据，这样便可在单个 SELECT 语句中使用 OPENROWSET()。
- BULK 选项的参数可对何时开始和结束数据读取、如何处理错误以及如何解释数据提供有效控制。例如，可以指定以类型为 varbinary、varchar 或 nvarchar 的单行单列行集的形式读取数据文件。默认行为详见随后的参数说明。

注意：当使用以完整恢复模式导入数据时，OPENROWSET（BULK...）不优化日志记录。

- 'data_file'：数据文件的完整路径，该文件的数据将被复制到目标表中。
- FORMATFILE='format_file_path'：指定格式化文件的完整路径。SQL Server 支持两种格式化文件类型：XML 和非 XML。格式化文件对定义结果集中的列类型是必需的。唯一的例外情况是指定 SINGLE_CLOB、SINGLE_BLOB 或 SINGLE_NCLOB 时，在这种情况下，不需要格式化文件。
- <bulk_options>：指定 BULK 选项的一个或多个参数。这些参数在 7.1.4 节中已经进行了介绍，此处就不再重复。
- SINGLE_BLOB：将 data_file 的内容作为类型为 varbinary（max）的单行单列行集返回。

🔔注意：建议仅使用 SINGLE_BLOB 选项（而不是 SINGLE_CLOB 和 SINGLE_NCLOB）导入 XML 数据，因为只有 SINGLE_BLOB 支持所有的 Windows 编码转换。

- SINGLE_CLOB：通过以 ASCII 格式读取 data_file，使用当前数据库的排序规则将内容作为类型为 varchar（max）的单行单列行集返回。
- SINGLE_NCLOB：通过以 UNICODE 格式读取 data_file，使用当前数据库的排序规则将内容作为类型为 nvarchar（max）的单行单列行集返回。

以 Excel 为例，先创建一个 Excel 文件 AddressType.xls，该文件中的内容如表 7.2 所示。

表 7.2　Excel中的数据

AddressTypeID	Name	rowguid	ModifiedDate
1	Billing	11	00:00.0
2	Home	22	00:00.0
3	Main Office	33	00:00.0

现在将 Excel 文件作为数据源，使用 OPENROWSET()函数访问该文件中的内容。具体脚本如代码 7.12 所示。

代码 7.12　使用 OPENROWSET()函数查询 Excel 表

```
SELECT *
FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',
'EXCEL 8.0;HDR=YES;IMEX=1;DATABASE=D:\AddressType.xls',
'select * from [Sheet1$]')
```

OPENROWSET 除了将 SQL 命令提交到远程服务器外，还可以将远程数据起一个别名，然后与其他表进行连接查询。例如，要将 Excel 表中的内容与 AdventureWorks 数据库中的 Person.AddressType 表进行内连接，则脚本如代码 7.13 所示。

代码 7.13　使用 OPENROWSET()函数与其他表连接查询

```
SELECT *
FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',
'EXCEL 8.0;HDR=YES;IMEX=1;DATABASE=D:\AddressType.xls',
'select * from [Sheet1$]') AS o
INNER JOIN AdventureWorks.Person.AddressType p
ON o.AddressTypeID=p.AddressTypeID
```

与 SELECT INTO 命令一起使用可以将远程数据源中的数据复制到数据库中。例如要将该 Excel 表中的内容复制到 TestDB1 数据库中，并将新表命名为 ATFromExcel，则对应的脚本如代码 7.14 所示。

代码 7.14　使用 OPENROWSET()函数复制数据到数据库

```
USE TestDB1;
GO
SELECT o.* INTO ATFromExcel
FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',
'EXCEL 8.0;HDR=YES;IMEX=1;DATABASE=D:\AddressType.xls',
```

```
'select * from [Sheet1$]') AS o
```

对于已经存在的表，若要复制数据到该表中，则应该使用 INSERT 命令。例如已经存在表 ATFromExcel，现在需要将 Excel 中的数据复制到该表中，则脚本如代码 7.15 所示。

代码 7.15　使用 OPENROWSET()函数复制数据到已有表中

```
USE TestDB1;
GO
INSERT INTO dbo.ATFromExcel
SELECT o.*
FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',
'EXCEL 8.0;HDR=YES;IMEX=1;DATABASE=D:\AddressType.xls',
'select * from [Sheet1$]') AS o
```

说明： OPENROWSET()函数除了能处理 Excel 外还可以处理 Oracle、Sybase、Access、文本文件等数据源，不同的数据源有不同的访问接口，访问 Oracle、Sybase 等非微软产品时需要单独安装数据库驱动才能获得访问接口。

7.2　复 制 概 述

在讲解了使用 bcp 进行批量数据处理后，可以使用 bcp 命令将远程数据源导入到本地数据库中，但是 bcp 命令在命令行下操作不够自动，而且不易管理。在自动的数据同步和批量操作上，SQL Server 中的复制（Replication）提供了强大的功能支持，本节将主要介绍复制的基础知识。

7.2.1　复制简介

复制是一组技术，它将数据和数据库对象从一个数据库复制和分发到另一个数据库，然后在数据库间进行同步，以维持一致性。可以在一个数据库实例中使用复制，也可以在局域网、广域网、拨号连接、无线连接和 Internet 上将数据分发到不同位置，以及分发给远程或移动用户。按照 SQL Server 的复制功能进行划分，有以下 3 类复制：事务复制、合并复制和快照复制。

- 事务复制通常用于需要高吞吐量的服务器到服务器方案（包括提高伸缩性和可用性、数据仓库和报告、集成多个站点的数据、集成异类数据，以及减轻批处理的负荷）。
- 合并复制主要是为可能存在数据冲突的移动应用程序或分布服务器应用程序设计的。常见应用场景包括：与移动用户交换数据、POS（消费者销售点）应用程序，以及集成来自多个站点的数据。
- 快照复制用于为事务复制和合并复制提供初始数据集；在适合数据完全刷新时也可以使用快照复制。

所有这些类型都依赖于 SQL Server 代理，以作业的方式运行，这些作业用来执行与跟踪更改和分发数据相关的任务。关于 SQL Server 代理的相关知识可以参见第 17 章的内容。利用这 3 种复制，SQL Server 提供功能强大且灵活的系统，以便使企业范围内的数据同步。

除了复制技术以外，还可以在 SQL Server 2012 中使用 Microsoft Sync Framework 和 Sync Services for ADO.NET 同步数据库。Sync Services for ADO.NET 提供了一个直观且灵活的 API，可用于生成面向脱机和协作应用场景的应用程序。

复制使用出版业的术语表示复制拓扑中的组件，其中有发布服务器、分发服务器、订阅服务器，另外还有发布、项目和订阅等术语。使用杂志术语有助于用户理解复制，复制的拓扑结构如图 7.5 所示。但是 SQL Server 复制包含有这套术语不能表述所有功能，对于订阅服务器进行更新的功能不能表述。另外，发布服务器将增量更改发送到发布中的项目的功能也不能表述。

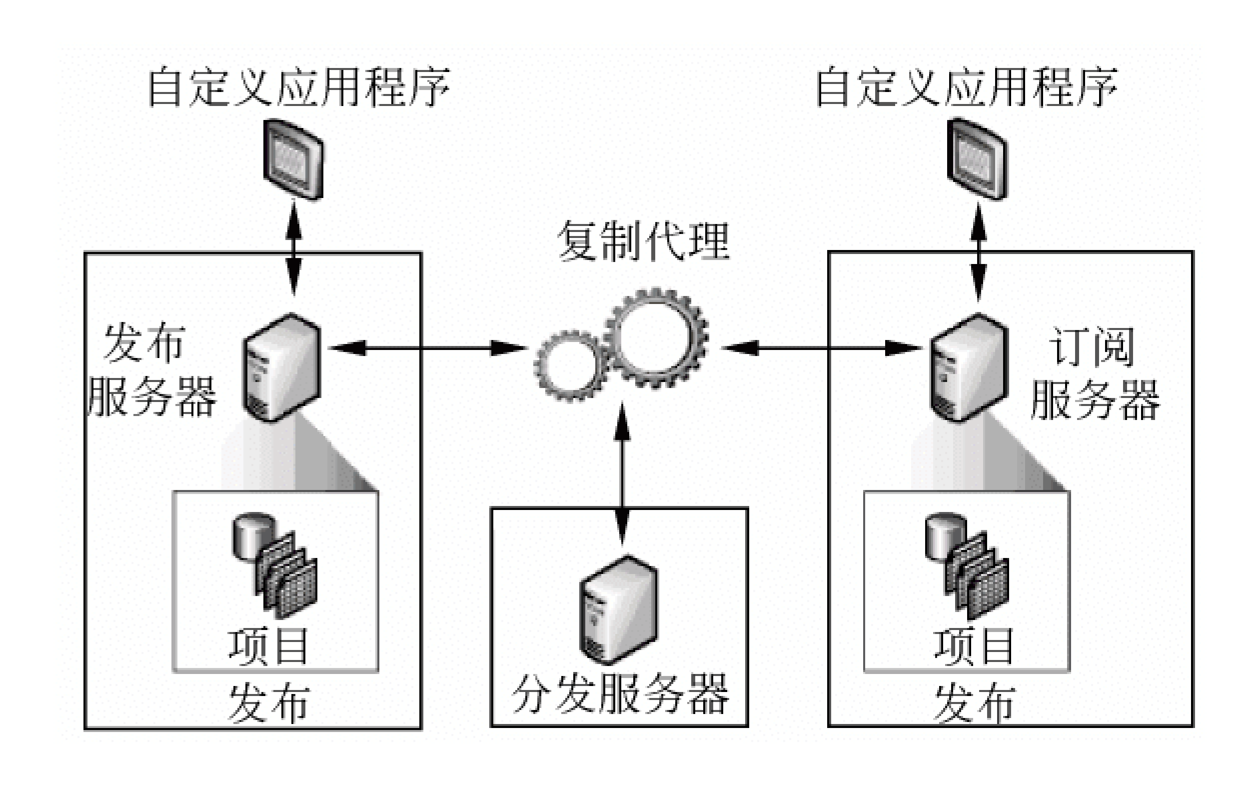


图 7.5　复制的拓扑结构

在 SQL Server 复制中，各服务器分别担任着重要的角色：

- 发布服务器在一种数据库实例中负责向其他位置提供数据。发布服务器可以有一个或多个发布，每个发布定义一组要复制的具有逻辑关系的对象和数据。
- 分发服务器是发布服务器与订阅服务器之间的桥梁，起着存储区的作用，负责复制与一个或多个发布服务器相关联的特定数据。每个发布服务器都与分发服务器上的单个数据库（称做分发数据库）相关联。分发数据库从发布服务器获得要发布的数据后将存储复制状态数据和有关发布的元数据，并且在某些情况下为从发布服务器向订阅服务器移动的数据起着排队的作用。在大多数情况下，一个数据库服务器实例充当发布服务器和分发服务器两个角色。当发布服务器和分发服务器在同一个数据库实例中时，称为“本地分发服务器”。当发布服务器和分发服务器按各自的数据库服务器实例配置时，把分发服务器称为“远程分发服务器”。
- 订阅服务器是从分发服务器接收复制数据的数据库实例。一个订阅服务器可以从多个发布服务器和发布中接收数据。根据所选复制的类型，如果是合并复制模式，订阅服务器还可以将数据更改传递回发布服务器，或者将数据重新发布到其他订阅服务器。

7.2.2　复制类型

前面已经讲到复制分为事务复制、合并复制和快照复制，这 3 种复制类型对应着不同的业务需求。

1．事务复制

事务性复制具有很高的一致性，当发布服务器中的数据发生更改时，订阅服务器上的数据也会很快得到更改。事务性复制通常从发布数据库对象和数据的快照开始。创建了初始快照后，接着在发布服务器上所做的数据更改和架构修改通常在修改发生时（几乎实时）便传递给订阅服务器。数据更改将按照其在发布服务器上发生的顺序和事务边界，应用于订阅服务器，因此，在发布内部可以保证事务的一致性。

事务性复制通常用于要求多个服务器之间的数据具有很高的一致性的情况，这种情况一般是服务器到服务器环境中。在以下各种情况下适合采用事务性复制：

- 希望发生增量更改时将其传播到订阅服务器。
- 从发布服务器上发生更改，至更改到达订阅服务器，应用程序需要这两者之间的滞后时间较短。
- 应用程序需要访问中间数据状态。
- 发布服务器有大量的插入、更新和删除活动。
- 发布服务器或订阅服务器不是 SQL Server 数据库（例如 Oracle）。

说明： *在系统情况下，事务性发布的订阅服务器应视为只读，即在订阅服务器上对数据进行更改将不会传播回发布服务器。但是，事务性复制确实提供了允许在订阅服务器上进行更新的选项。*

2．合并复制

合并复制通常也是从发布数据库对象和数据的快照开始，并且用触发器跟踪在发布服务器和订阅服务器上所做的后续数据更改和架构修改。合并复制情况下允许对发布服务器和定义服务器中的数据进行更改，订阅服务器并不一直与发布服务器保持连接，只有订阅服务器在连接到网络时才与发布服务器进行同步，系统将自动合并自上次同步以来发布服务器和订阅服务器之间发生更改的所有行。

合并复制通常用于服务器到客户端的环境中，特别是移动数据库与服务器数据库之间。合并复制适用于下列几种情况：

- 多个订阅服务器可能会在不同时间更新同一数据，并将其更改传播到发布服务器和其他订阅服务器。
- 订阅服务器需要接收数据，脱机更改数据，并在以后与发布服务器和其他订阅服务器同步更改。
- 每个订阅服务器都需要不同的数据分区。
- 可能会发生冲突，并且在冲突发生时，用户需要具有检测和解决冲突的能力。
- 应用程序需要最终的数据更改结果，而不是访问中间数据状态。

合并复制允许在每个订阅服务器上对单独对数据进行更改，并在以后将更新合并成一个统一的结果。由于更新是在多个节点上进行的，所以可能出现同一数据在发布服务器和多个订阅服务器上进行了不同操作的更新。因此，在合并更新时可能会产生冲突，合并复制提供了多种处理冲突的方法。

3．快照复制

快照复制与前面提到的数据库快照技术类似，是将数据以特定时刻的瞬时状态分发，而不监视对数据的更新。在数据同步时，系统将重新生成完整的快照并将其发送到订阅服务器。

当符合以下一个或多个条件时，使用快照复制本身是最合适的：

- 很少更改数据。
- 在一段时间内允许具有相对发布服务器已过时的数据副本。
- 复制少量数据。
- 在短期内出现大量更改。

与事务复制相反，快照复制中服务器之间并不具有很高的一致性，快照复制是按照作业计划或者人工操作进行数据同步的，在数据同步之前发布服务器上的数据与订阅服务器上的数据可能存在很大的差别。在数据更改量很大，但很少发生时，快照复制是最合适的。

快照复制常用于多个系统中作为数据接口，例如，一个企业的多个系统都要使用部门人员组织结构数据库，每个系统中使用了这些数据，由于组织结构并不是经常变化，所以可以对该数据库使用快照复制，在更改了发布服务器上的组织结构后等到更新复制时才将数据更新到每一个订阅服务器上。

发布服务器上快照复制的连续开销低于事务性复制的开销，因为不用跟踪增量更改。但是，如果要复制的数据集非常大，由于每次同步数据时都是重新生成快照并将所有数据发送给订阅服务器，所以对于很大的数据集，在快照复制中将需要使用大量资源。需要考虑整个数据集的大小，以及数据的更改频率以决定是否使用快照复制。

7.2.3　复制代理

前面已经提到，复制使用许多称为 SQL Server 代理的独立程序执行与跟踪更改和分发数据关联的任务。在 SQL Server 2012 中，复制代理作为 SQL Server 代理安排的作业运行，必须运行 SQL Server 代理，这些作业才能运行。复制代理本质上也是普通的一个 SQL Server 代理，所以像一般的 SQL Server 代理一样，复制代理可以从命令行以及由使用复制管理对象（RMO）的应用程序运行。可以使用 SQL Server 复制监视器和 SSMS 等工具对复制代理进行管理。

注意：SQL Server 代理作为一个独立的应用程序并不随 SQL Server 服务的启动而启动，可以通过 Windows 服务管理器或者 SQL Server 配置管理器将 SQL Server 代理设置为自启动，这样每次开机时系统将自动启动 SQL Server 代理。

复制代理按照功能不同分为以下几种类型：

- 快照代理在各种类型的复制中都使用。快照代理主要负责准备已发布表的架构和

初始数据文件以及其他对象、存储快照文件并记录分发数据库中的同步信息。快照代理是运行在分发服务器上的。

- 日志读取器代理在事务性复制中使用。日志读取器代理负责将发布服务器上的事务日志中标记为复制的事务移至分发数据库中。使用事务性复制发布的每个数据库都有自己的日志读取器代理，该代理运行于分发服务器上并与发布服务器连接。
- 分发代理在快照复制和事务性复制中使用。分发代理负责将初始快照应用于订阅服务器，并将分发数据库中保存的事务移至订阅服务器。推送订阅中分发代理运行于分发服务器上，而在请求订阅中则可运行于订阅服务器上。
- 合并代理与合并复制一起使用。合并代理负责将初始快照应用于订阅服务器，并移动和协调所发生的增量数据更改。在每个合并订阅上都有自己的合并代理，该代理同时连接到发布服务器和订阅服务器并对数据进行更新。与分发代理类似，合并代理既可以运行于分发服务器（对于推送订阅），也可以运行于订阅服务器（对于请求订阅）。默认情况下，合并代理将订阅服务器上的数据更改上传到发布服务器，然后将发布服务器上的更改下载到订阅服务器。
- 队列读取器代理是在包含排队更新选项的事务性复制中使用。队列读取器代理运行于分发服务器上，负责将订阅服务器上所做的更改移回至发布服务器。与分发代理和合并代理不同，只有一个队列读取器代理的实例为给定分发数据库的所有发布服务器和发布提供服务。
- 复制包含许多执行计划维护和按需维护的维护作业。

7.2.4　订阅简介

订阅是对发布中的数据和数据库对象副本的请求。订阅定义将接收哪个发布，以及接收的时间和位置。订阅分为推送订阅和请求订阅，在计划订阅时，需要考虑代理处理发生的位置决定使用的订阅类型，所选择的订阅类型将控制代理运行的位置。

对于推送订阅，合并代理或分发代理在分发服务器上运行，发布服务器将更改传播到订阅服务器，而无需订阅服务器发出请求。更改可以按需、连续地或按照计划推送到订阅服务器。分发代理或合并代理在分发服务器上运行。在以下情况时可以使用推送订阅：

- 数据将连续同步或按照经常重复执行的计划同步。
- 发布要求数据近似实时地移动。
- 分发服务器上较高的处理器开销不会影响性能。
- 通常与快照和事务复制一起使用。

对于请求订阅，代理在订阅服务器上运行，订阅服务器请求在发布服务器上所做的更改。请求订阅允许订阅服务器上的用户确定同步数据更改的时间。分发代理或合并代理在订阅服务器上运行。在以下情况下可以使用请求订阅：

- 数据通常按需或按计划同步，而非连续同步。
- 发布具有大量订阅服务器，并且（或）在分发服务器上运行所有代理会消耗大量资源。
- 订阅服务器是自主的、断开连接的和（或）移动的。 订阅服务器将确定连接和同

步更改的时间。

- 通常与合并复制一起使用。

注意：创建订阅后，将无法更改其类型。

所有复制类型都允许推送订阅和请求订阅。合并复制中又分为客户端订阅和服务器订阅。客户端订阅和服务器订阅类型都可用于推送订阅和请求订阅。客户端订阅适合于大多数订阅服务器，而服务器订阅通常用于向其他订阅服务器重新发布数据的订阅服务器。订阅选择还会影响冲突解决。

7.3　复制的工作机制

在了解了复制的基本概念后，本节主要从复制的本质出发，介绍每种类型的复制工作机制，了解复制的原理。

7.3.1　快照复制工作机制

无论是快照复制、事务复制还是合并复制，这 3 种复制都使用快照初始化订阅服务器。SQL Server 快照代理负责生成快照文件，但传递文件的代理因使用的复制类型而异。快照复制和事务性复制使用分发代理传递文件，而合并复制使用 SQL Server 合并代理。快照代理在分发服务器上运行。推送订阅模式下分发代理和合并代理在分发服务器上运行；而对于请求订阅，则在订阅服务器上运行。快照复制的工作机制如图 7.6 所示。

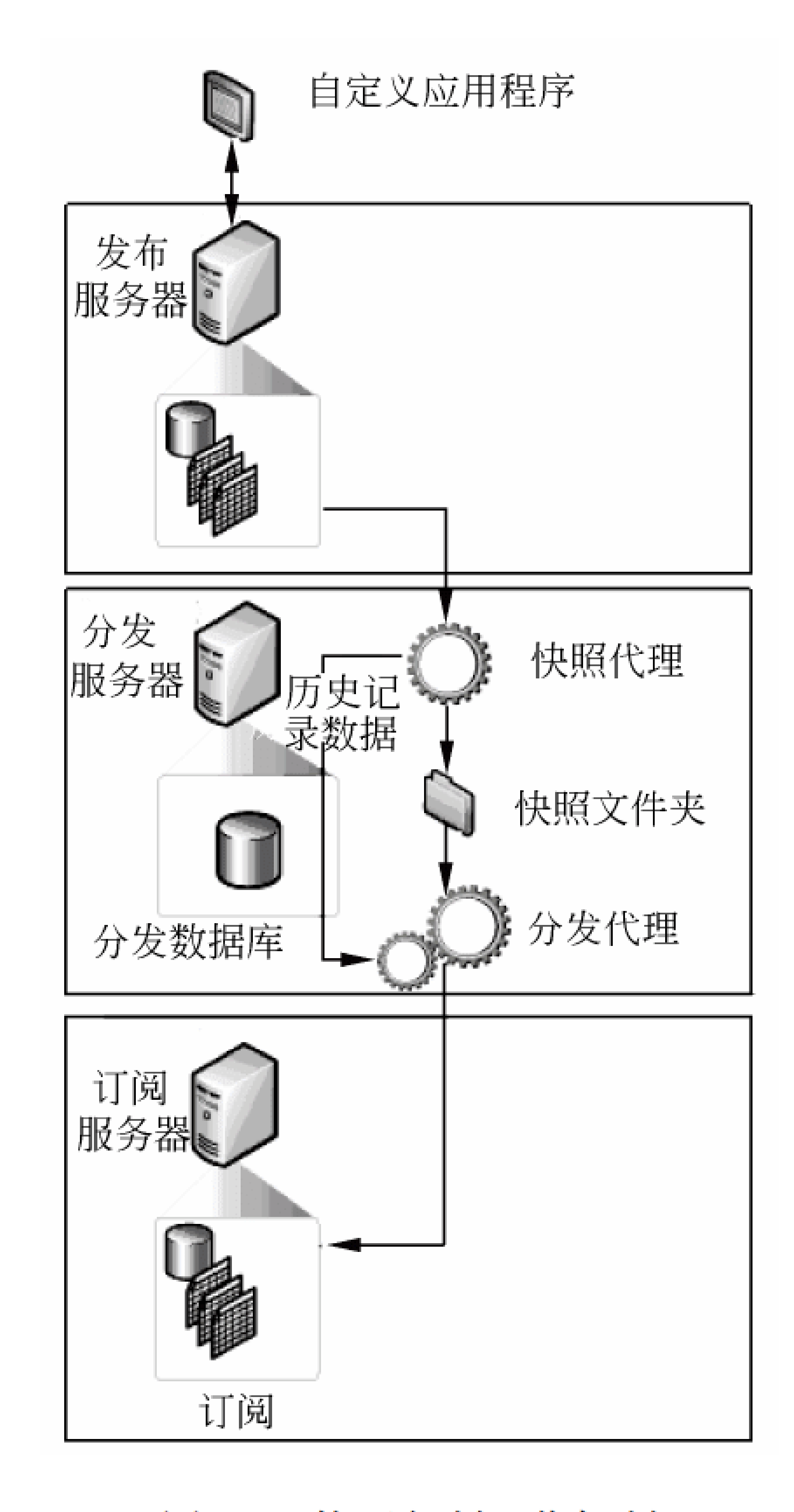


图 7.6　快照复制工作机制

从图 7.6 中可以看出，当快照复制运行时，在发布服务器上快照代理执行了以下操作：

（1）在分发服务器到发布服务器之间建立连接，然后根据需要在已发布表上使用锁：

- 对于合并发布，快照代理不使用任何锁。
- 对于事务性发布，默认情况下快照代理只在快照生成的初始阶段使用锁。
- 对于快照发布，整个快照生成过程中都使用锁。

（2）快照复制将每个项目的表架构副本写入一个.sch 文件。如果发布其他数据库对象（如索引、约束、存储过程、视图、用户定义函数等），则生成其他脚本文件。

（3）从发布服务器上已发布的表中复制数据，然后将数据写入快照文件夹。快照数据文件将以一组 BCP 导出文件的形式生成。

（4）对于快照发布和事务性发布，快照代理将向分发数据库的 MSrepl_commands 表和 MSrepl_transactions 表中追加行。MSrepl_commands 表中包含的是命令，这些命令指示.sch 和.bcp 文件、其他快照文件，以及对快照前或快照后脚本的引用位置。MSrepl_transactions 表中包含的是与同步订阅服务器相关的命令。

（5）释放已发布表上的所有锁。

快照文件生成后，接下来由分发代理执行分发工作：

（1）与分发服务器建立连接。

（2）检查分发服务器上分发数据库中的 MSrepl_commands 和 MSrepl_transactions 表。代理从第一个表中读取快照文件的位置，并从这两个表中读取订阅服务器同步命令。

（3）复制快照中的数据，将架构和命令应用于订阅数据库。

说明：如果所有订阅服务器都是 SQL Server，那么快照数据就被存储为本地 bcp 文件；否则将会创建字符模型文件以支持不同的订阅服务器。

7.3.2　事务复制工作机制

事务性复制由 SQL Server 快照代理、日志读取器代理和分发代理实现。同样还是由快照代理准备快照文件，快照文件中包含了已发布表和数据库对象的架构和数据。然后将这些文件存储在快照文件夹中，并在分发服务器上的分发数据库中记录同步作业。

分发数据库的作用相当于一个可靠的存储转发队列，日志读取器代理监视为事务性复制配置的每个数据库的事务日志，并将标记为要复制的事务从事务日志复制到分发数据库中。分发代理将快照文件夹中的初始快照文件和分发数据库表中的事务复制到订阅服务器上。

在发布服务器上，对数据库进行的数据更改系统将把增量更改信息根据分发代理的计划流向订阅服务器，分发代理保存连续运行以尽量减少滞后时间，也可以按预定的时间间隔运行。在事务复制模式下，订阅数据库相当于只读的，数据更改必须在发布服务器上进行，从而避免了更新冲突。最后，所有订阅服务器都将获得与发布服务器相同的值。如果事务性复制使用了立即更新或排队更新选项，则更新可以在订阅服务器上进行，对于排队更新，可能会发生冲突。整个事务复制的工作原理如图 7.7 所示。

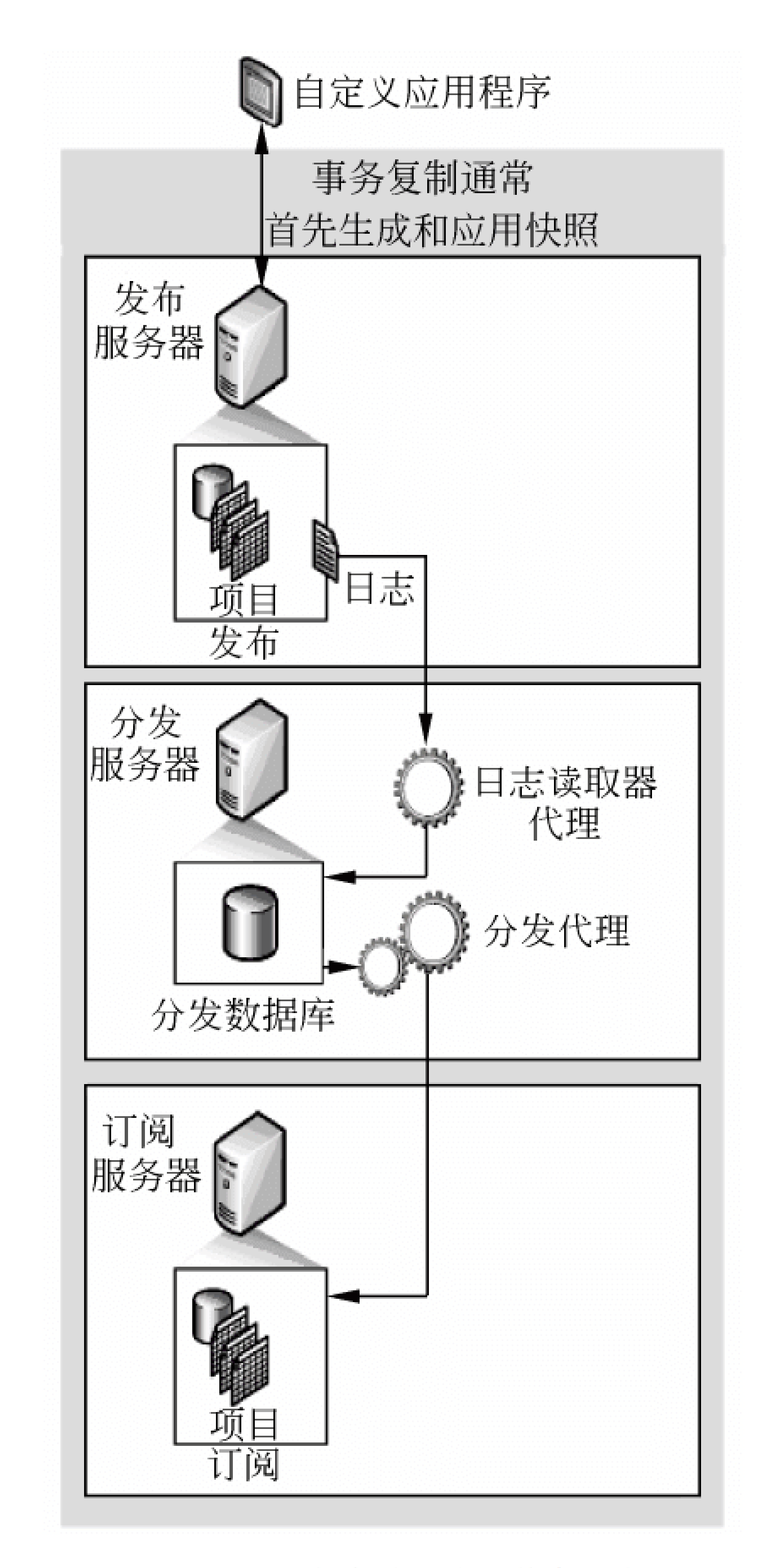


图 7.7　事务复制工作机制

从图 7.7 中可以看出，事务复制的工作过程为：

（1）在进行新的事务复制之前需要初始化数据，新的事务性复制订阅服务器上必须包含一些表，这些表需要与发布服务器上的表具有相同的架构和数据，这样才能从发布服务器上接收增量更改。初始数据集通常是由

快照代理创建并由分发代理分发和应用的快照。初始数据集还可以通过备份或其他方式提供。

（2）日志读取器代理在分发服务器上运行，通常连续运行，但也可以按照用户制订的计划运行。执行日志读取器代理时，代理首先读取发布事务日志，并标识任何 INSERT、UPDATE，以及 DELETE 语句，或者对已标记为要复制的事务进行其他数据修改。然后，该代理将这些事务批量复制到分发服务器上的分发数据库中。日志读取器代理使用内部存储过程 sp_replcmds 从日志中获取标记为要复制的下一个命令集。这样，分发数据库就成为一个存储转发队列，从该队列中将更改发送到订阅服务器上。

注意：只有已经提交的事务才会被发送到分发数据库中，未提交或回滚的事务将不会发送。

（3）当整批事务都成功写入分发数据库之后，将提交这批事务。在每一批命令都提交到分发服务器后，日志读取器代理将调用 sp_repldone 以标记最终完成复制的位置。最后，代理在事务日志中标记可以清除的行。仍在等待复制的行不会被清除。事务命令在传播到所有订阅服务器或达到最大分发保持期之前，一直存储在分发数据库中。订阅服务器按事务在发布服务器上应用的相同顺序接收事务。

注意：前面讲到分发代理在分发服务器（对于推送订阅）和订阅服务器（对于请求订阅）上运行。该代理负责将事务从分发数据库移动到订阅服务器上。如果订阅被标记为需要验证，则分发代理还要检查发布服务器和订阅服务器上的数据是否匹配。

7.3.3　Oracle 发布工作机制

Oracle 发布也分为快照复制和事务复制两种，其实现的工作机制与 SQL Server 数据库上的快照复制和事务复制并没有太大的不同。

从 SQL Server 2005 起支持 Oracle 发布，Oracle 快照发布与 SQL Server 快照发布在实现方式上类似。如果运行 Oracle 快照发布，分发服务器上的快照代理将连接到 Oracle 发布服务器，并处理发布中的每个表。处理每个表时，代理将检索表行并创建架构脚本，然后将该脚本存储在发布的快照共享中。每次运行快照代理时都会创建整个数据集，因此在事务性复制中出现更改跟踪触发器时，不会将它们添加到 Oracle 表中。快照复制是迁移数据的一种便利方法，可以将对发布系统的影响降到最低。

Oracle 事务性发布的实现与 SQL Server 的事务性发布体系结构相似，但更改却是通过结合使用 Oracle 数据库上的数据库触发器和日志读取器代理来跟踪的。Oracle 事务性发布的订阅服务器也是通过快照复制自动进行初始化。在以后发生更改时，将通过日志读取器代理对这些更改进行跟踪，并将更改传递到订阅服务器。

在创建 Oracle 发布时，系统将为 Oracle 数据库中所有已发布的表创建触发器和跟踪表。对发布的表进行数据更改时，将激发这些表的数据库触发器，并在复制跟踪表中为修改的每一行插入信息。然后，SQL Server 分发服务器上的日志读取器代理将数据更改信息从跟踪表移动到分发服务器上的分发数据库中。最后，与在标准事务性复制中一样，分发代理将更改从分发服务器移动到订阅服务器。

在 Oracle 事务复制中更改数据时，系统将执行以下步骤。

（1）用户或应用程序在一个或多个为复制而发布的 Oracle 表上执行插入、更新或删除操作。

（2）在 Oracle 服务器中修改的每一行，都会激发由复制在每个已发布的 Oracle 表上安装的行级触发器，触发器将有关更改的信息存储在与该表关联的项目日志表中。

（3）在修改数据激发行级触发器时，系统将从 HREPL_seq 序列中检索一个数字并将其分配给描述 DML 操作的日志表行。这样可以确保复制按照正确的顺序在订阅服务器上应用更改命令。

（4）如果修改数据时发生主键更新，还将激发安装在表上的语句级触发器，使同一语句中发生的多个主键更新相互关联。语句标识符从 HREPL_Stmt 序列中提取。此标识符用于正确处理订阅服务器上的主键更新。

（5）对于在发布的 Oracle 表中插入或删除的每一行，都会在关联的项目日志表中插入一行。对于 Oracle 表中更新的每一行，会在日志表中插入一行（后像）或两行（前像和后像），具体情况取决于复制是否需要该行以前的状态信息。

而在分发服务器上，SQL Server 系统执行了以下操作：

（1）首先是日志读取器代理在项目日志中标识出尚未同步的数据更改，并将日志表项的行 ID 暂时存储在 HREPL_Poll 表中。

（2）从序列 HREPL_Pollid 中提取的标识符用于将每个更改项标记为事务一致性组的成员，并提供该组相对于其他组的处理顺序。

（3）代理在处理已发布表的更改时，将从日志表中检索行，还要使用 HREPL_Pollid 中的当前轮询 ID 标识要处理的行。

（4）在分发数据库中日志表中的更改将作为一个事务提交，并将事务存储在复制相关的 MSrepl_commands 表和 MSrepl_transactions 表中。

（5）分发代理从分发数据库中读取更改并将更改传递给订阅服务器，这与 SQL Server 中的标准事务性复制相同。

7.3.4　合并复制工作机制

合并复制是由 SQL Server 快照代理和合并代理两个程序实现的。合并代理将初始快照应用于订阅服务器。代理还将合并自初始快照创建后发布服务器或订阅服务器上所发生的增量数据更改，并根据所配置的规则检测和解决任何冲突。合并复制的流程和机制如图 7.8 所示。

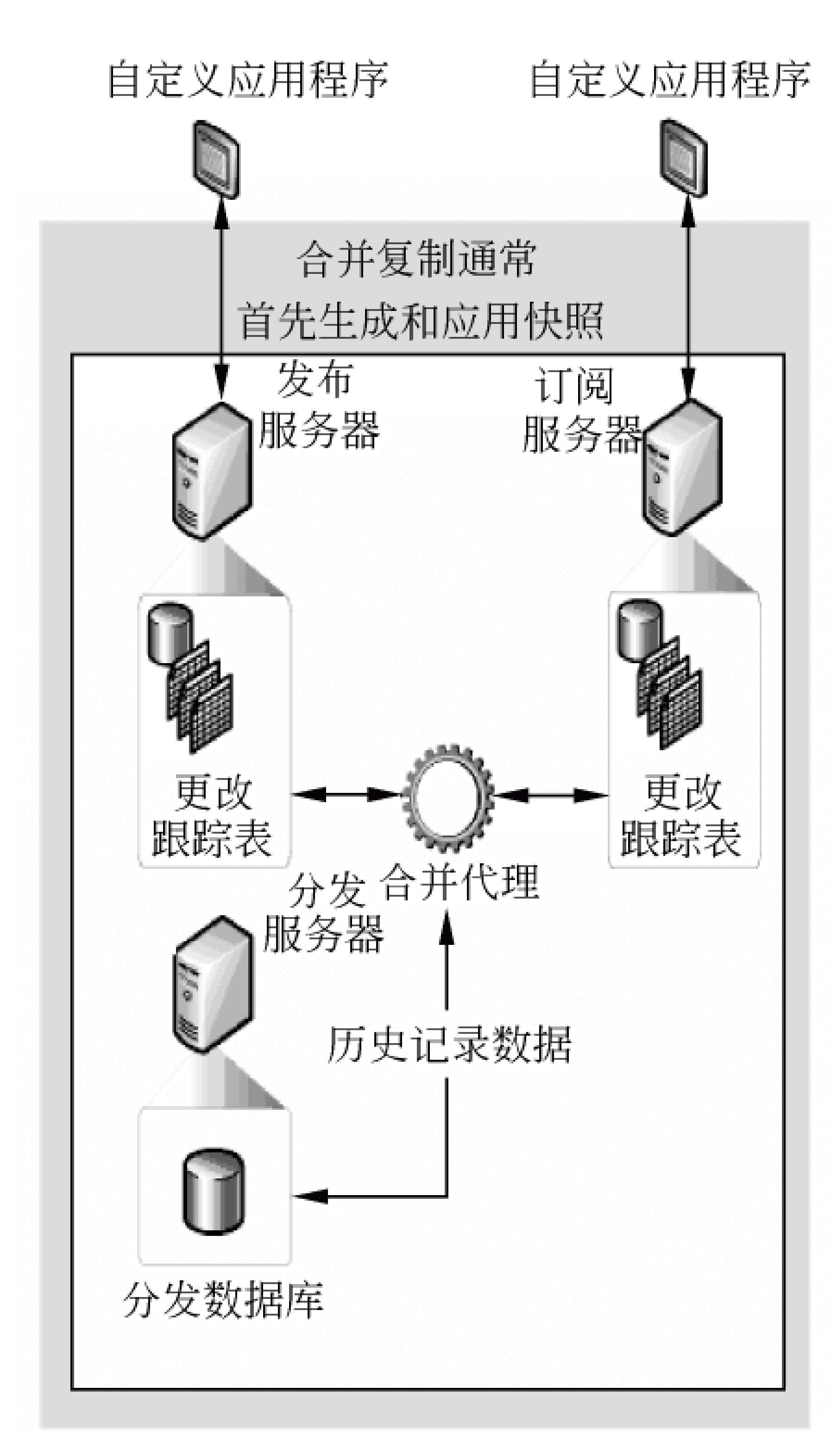


图 7.8　合并复制

合并复制运行之前必须先初始化发布服务器和订阅服务器，然后才能在它们之间传递数据。

在初始化发布或订阅后，合并复制将跟踪已发布表中数据的所有更改。当数据更改时，通过触发

器（由复制为每个已发布的表创建）和发布及订阅数据库中的系统表跟踪更改。这些复制系统表用指示应传播更改的元数据来填充。合并代理在同步过程中运行时，代理枚举所有更改，然后根据需要将更改应用到发布服务器和订阅服务器。

当使用了参数化筛选器和联接筛选器筛选合并发布中的一个或多个表后，已发布表中的数据将被分区。“分区”只是表中的部分行。订阅服务器与发布服务器同步时，发布服务器必须根据订阅服务器提供给系统函数 SUSER_SNAME()及 HOST_NAME()的值，确定哪些行属于订阅服务器的分区。

当合并代理在同步期间枚举要应用的更改时，会比较发布服务器和订阅服务器上每一行的元数据。合并代理用此元数据确定行或列是否在拓扑中多个节点上进行了更改，而这将指示出潜在的冲突。系统检测到冲突后，合并代理启动为发生冲突的项目所指定的冲突解决程序，并用此解决程序确定冲突解决入选方。入选行将应用到发布服务器和订阅服务器，而落选行的数据则写入一个冲突表中。

如果订阅过期，则必须对其重新初始化，因为该订阅的元数据已被删除。未重新初始化的订阅将由发布服务器上运行的“过期订阅清除”作业删除。默认情况下，此作业每天运行；作业负责删除在两倍的发布保持期内尚未同步的所有推送订阅。

注意：合并触发器不好干扰用户定义的触发器的位置和使用方式。

7.4　配 置 复 制

配置复制分为配置发布服务器、配置分发服务器和配置订阅服务器 3 种，其与 SQL Server 的配置方法基本相同，都是通过向导来完成，本节主要以快照复制的配置为例，讲解复制的配置过程。快照复制中一般将发布和分发配置在同一台服务器上从而减少了配置操作、软硬件成本和维护成本。

7.4.1　准备用于复制的服务器

在配置复制之前需要为数据库服务器配置账户安全和共享文件夹等信息。具体准备工作如下所述。

1. 创建Windows账户以运行复制代理

将在本地服务器上的代理创建一个单独的 Windows 账户。具体代理名称运行的位置和对应的账户名如表 7.3 所示。

表 7.3　运行代理的账户名

代　　理	位　　置	账　户　名
快照代理	发布服务器	<machine_name>\repl_snapshot
日志读取器代理	发布服务器	<machine_name>\repl_logreader
分发代理	发布服务器和订阅服务器	<machine_name>\repl_distribution
合并代理	发布服务器和订阅服务器	<machine_name>\repl_merge

在 Windows 上创建账户的操作为：

（1）在发布服务器上，从“控制面板”的“管理工具”中打开“计算机管理”选项。

（2）在“系统工具”中，展开“本地用户和组”。

（3）右击“用户”命令，在弹出的快捷菜单中选择“新建用户”选项。

（4）在“用户名”文本框中，输入 repl_snapshot，提供密码和其他相关信息，然后单击“创建”按钮来创建 repl_snapshot 账户。

（5）在订阅服务器上重复上述步骤创建 repl_distribution 账户。

2. 准备快照文件夹

接下来需要配置用于创建和存储发布快照的快照文件夹。具体创建快照文件夹的过程如下所述。

（1）新建文件夹 C:\Share，用于存储快照文件。

（2）右击该文件夹，在弹出的快捷菜单中选择“共享和安全”选项。

（3）选择“共享”选项卡，选择“共享此文件夹”单选框。 确保“共享名”的值 Share。

（4）单击“权限”按钮，弹出权限设置对话框。

（5）单击“添加”按钮。在“输入对象名称来选择”文本框中，输入前面创建的快照代理账户的名称，格式为<Machine_Name>\repl_snapshot，其中<Machine_Name>是发布服务器的名称。然后单击“确定”按钮。

（6）设置 repl_snapshot 用户允许完全控制、更改和读取该文件夹。

（7）同样的方法设置分发代理对该文件夹有读取权限。

（8）最后单击“确定”按钮，完成 C:\Share 文件夹的共享和权限配置工作。

3. 配置发布数据库权限

在 Windows 上建立了账户并添加了对快照文件夹的访问权限后，需要将该账户添加到数据库中。具体配置方法如下所述。

（1）在 SSMS 的对象资源管理器中，展开“安全性”节点，右击“登录名”节点，然后选择“新建登录名”选项。

（2）在“常规”页中单击“搜索”，在“输入要选择的对象名称”框中输入<Machine_Name>\repl_snapshot（其中，<Machine_Name>是本地发布服务器的名称），然后单击“确定”按钮。

（3）在“用户映射”页中，启用到 distribution 数据库和 AdventureWorks2012 数据库的用户映射，并向这些数据库的 db_owner 数据库角色成员身份添加用户。

（4）单击“确定”按钮创建登录名。

（5）用同样的方法为订阅服务器创建数据库权限。

7.4.2　配置快照发布和分发

配置发布可以通过“新建发布向导”来完成，以快照发布 AdventureWorks2012 数据库为例，将发布功能和分发功能配置在同一台服务器上，具体配置过程如下所述。

（1）打开 SSMS，在对象资源管理器中展开“复制”节点下的“本地发布”节点。

注意：快照的配置必须使用 SSMS 连接实际的服务器名称，不能通过服务器别名、IP 地址或者任何其他备用名称进行连接。一般是使用远程桌面连接到数据库服务器上，在服务器上使用 SSMS 进行配置。

（2）右击“本地发布”节点，在弹出的快捷菜单中选择“新建发布”选项，系统弹出“新建发布向导”对话框。

（3）单击“下一步”按钮，系统进入配置分发服务器界面，默认选中将充当自己的分发服务器，如图 7.9 所示。

（4）由于这里需要的就是将发布和分发配置在该数据库上，所以此处选中默认配置不变，单击“下一步”按钮进入快照文件夹对话框，如图 7.10 所示。



图 7.9　选择分发服务器

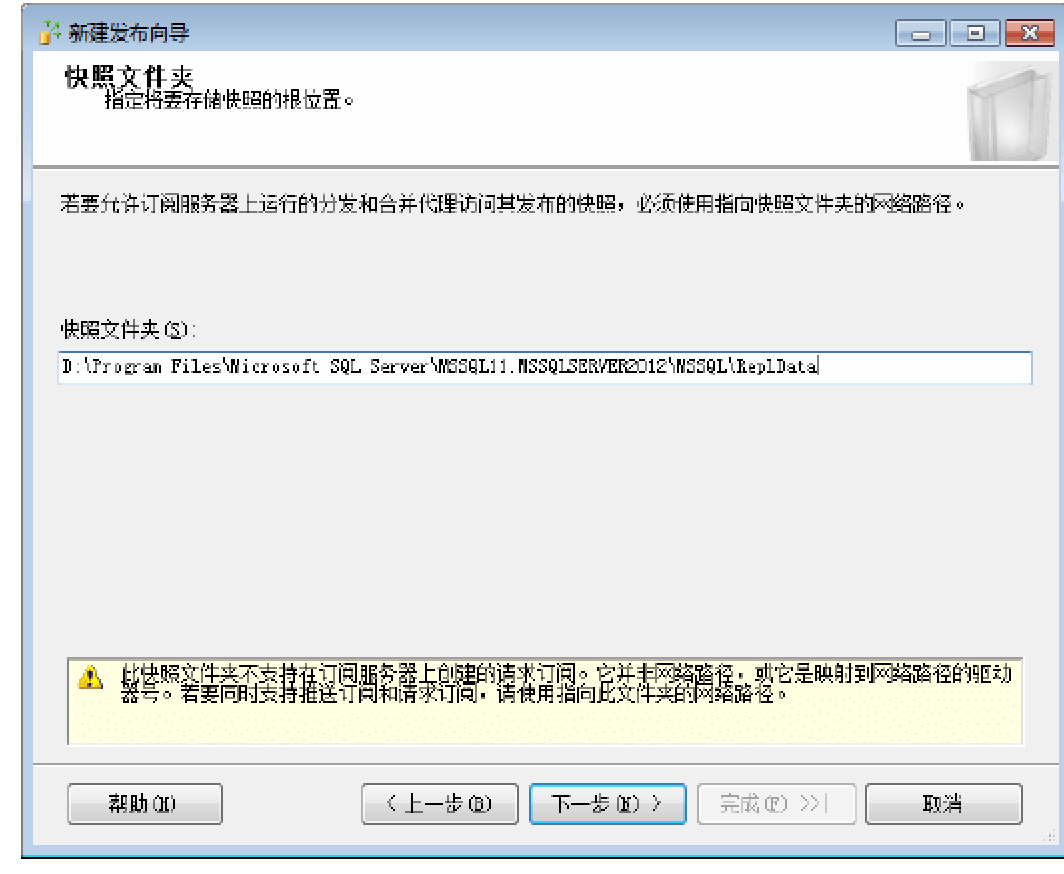


图 7.10　配置快照文件夹

（5）在“快照文件夹”文本框中输入\\192.168.100.4\share，然后单击“下一步”按钮进入“发布数据库”对话框，如图 7.11 所示。

（6）由于需要发布的数据库是 AdventureWorks2012，所以选择该数据库，然后单击“下一步”按钮，系统进入“发布类型”对话框，如图 7.12 所示。

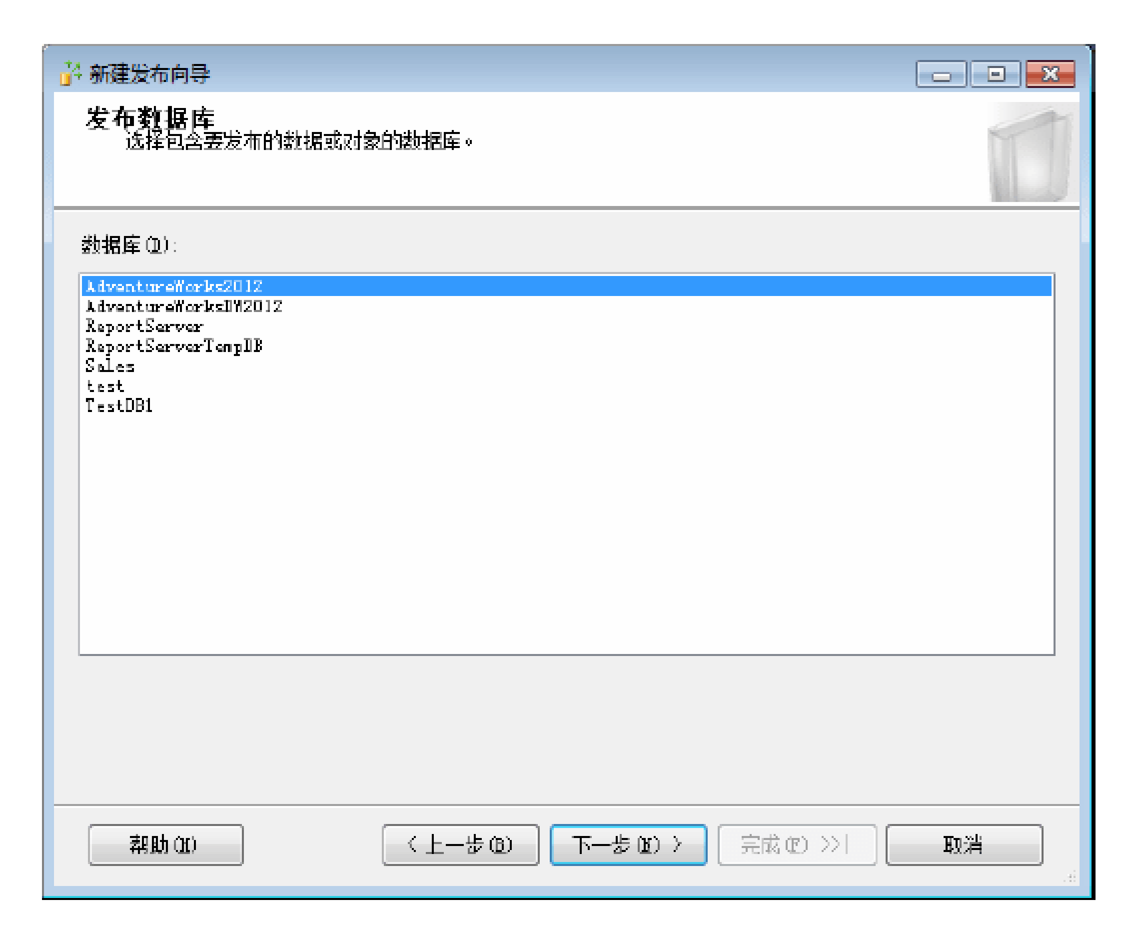


图 7.11　“发布数据库”对话框

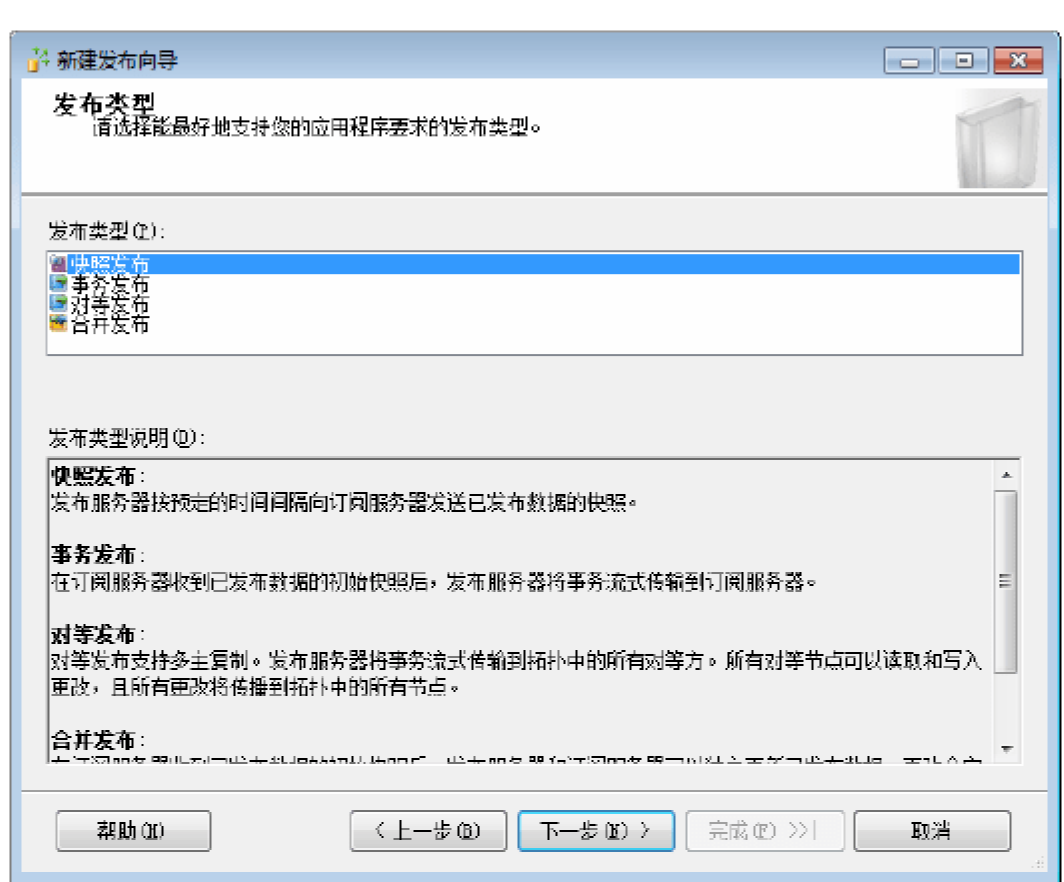


图 7.12　“发布类型”对话框

（7）由于这里是要配置快照发布，所以选择“快照发布”选项，然后单击“下一步”按钮，系统进入“项目”对话框，如图 7.13 所示。

（8）发布向导列出了当前用户所有的能够用于发布的数据库对象。由于本次发布只是一个示例，所以只选择两个表即可，实际发布中需要根据需求来选择要发布的对象。假设要发布表 Address 和 AddressType，则展开表节点，选中这两个表，如图 7.14 所示。

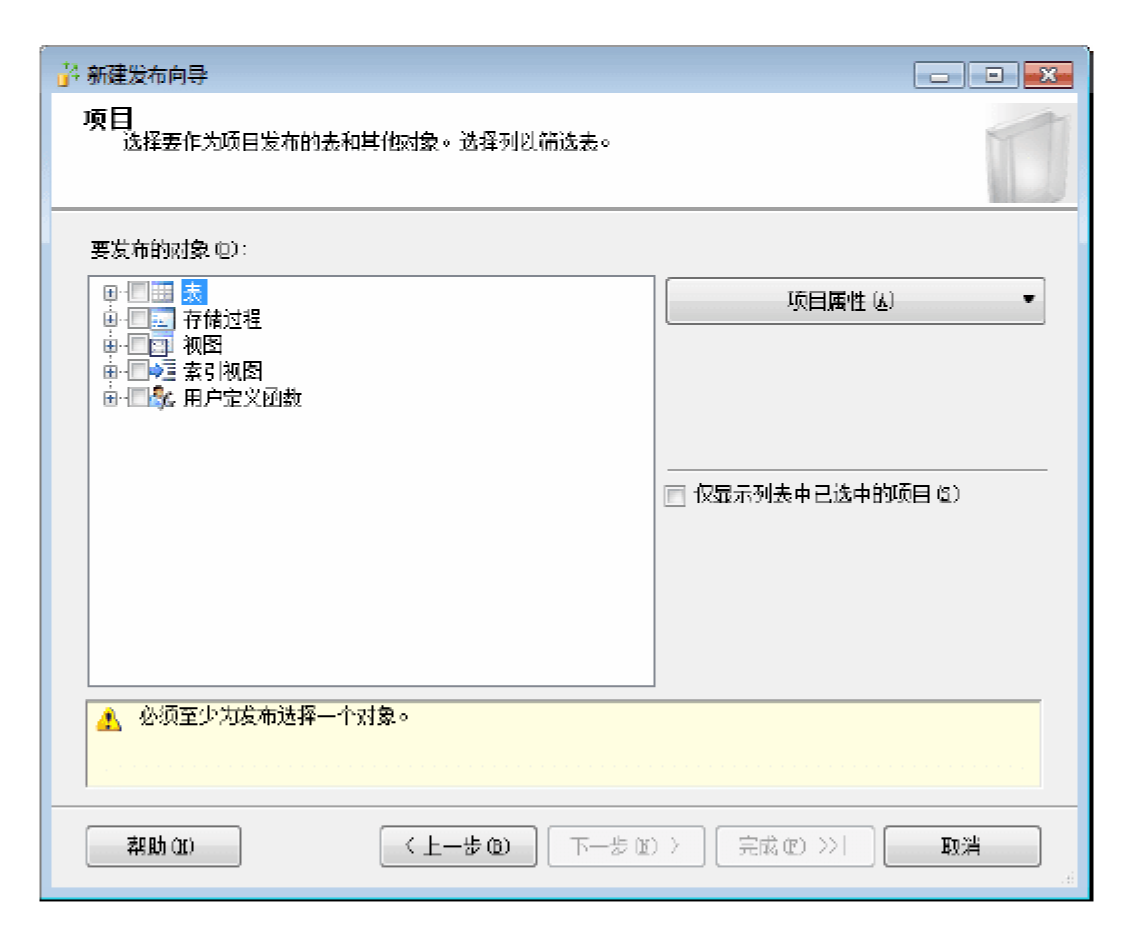


图 7.13　选择要发布的对象

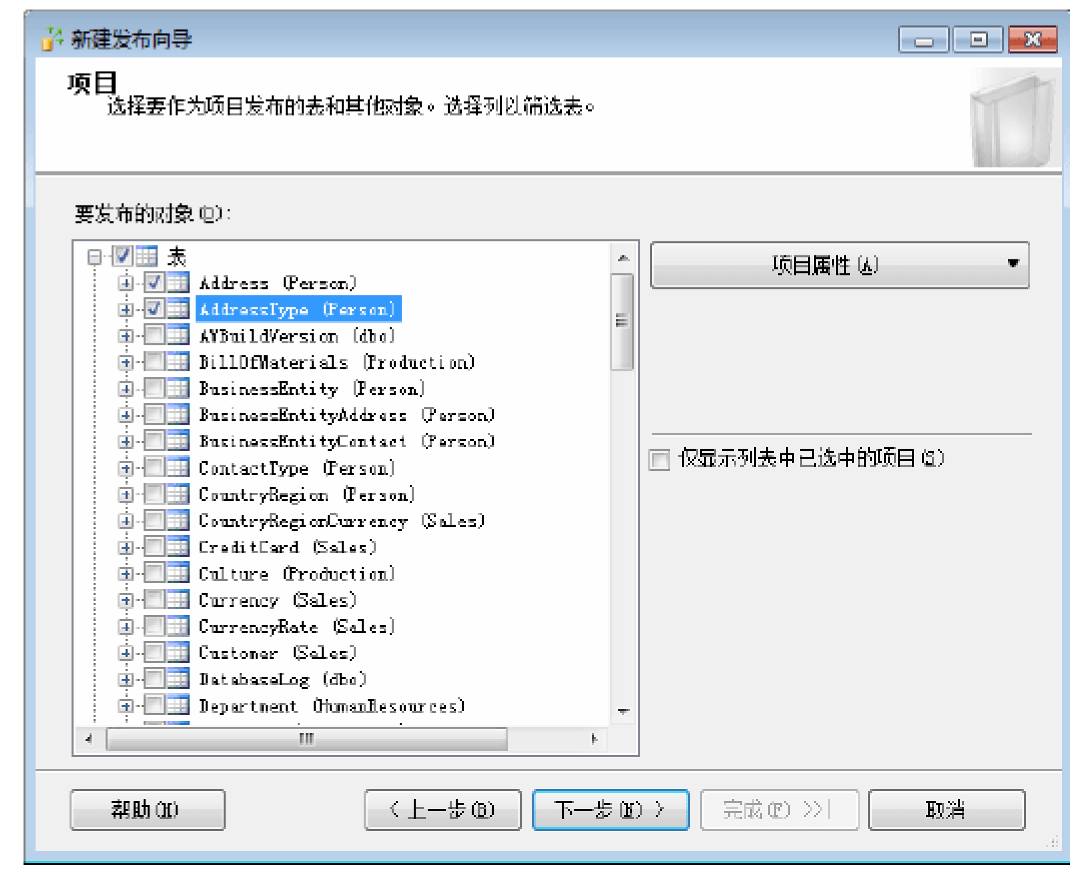


图 7.14　选中要发布的数据库对象

（9）若要在发布时同时发布表的约束、索引等，则选中要发布的表，然后单击“项目属性”按钮，在下拉列表中选择“设置突出显示的表项目的属性”选项，系统弹出选中表的项目属性对话框，如图 7.15 所示。

（10）根据实际项目需要设置具体项目属性，然后单击“确定”按钮回到发布向导对话框。在其中单击“下一步”按钮，进入“筛选表行”对话框，如图 7.16 所示。

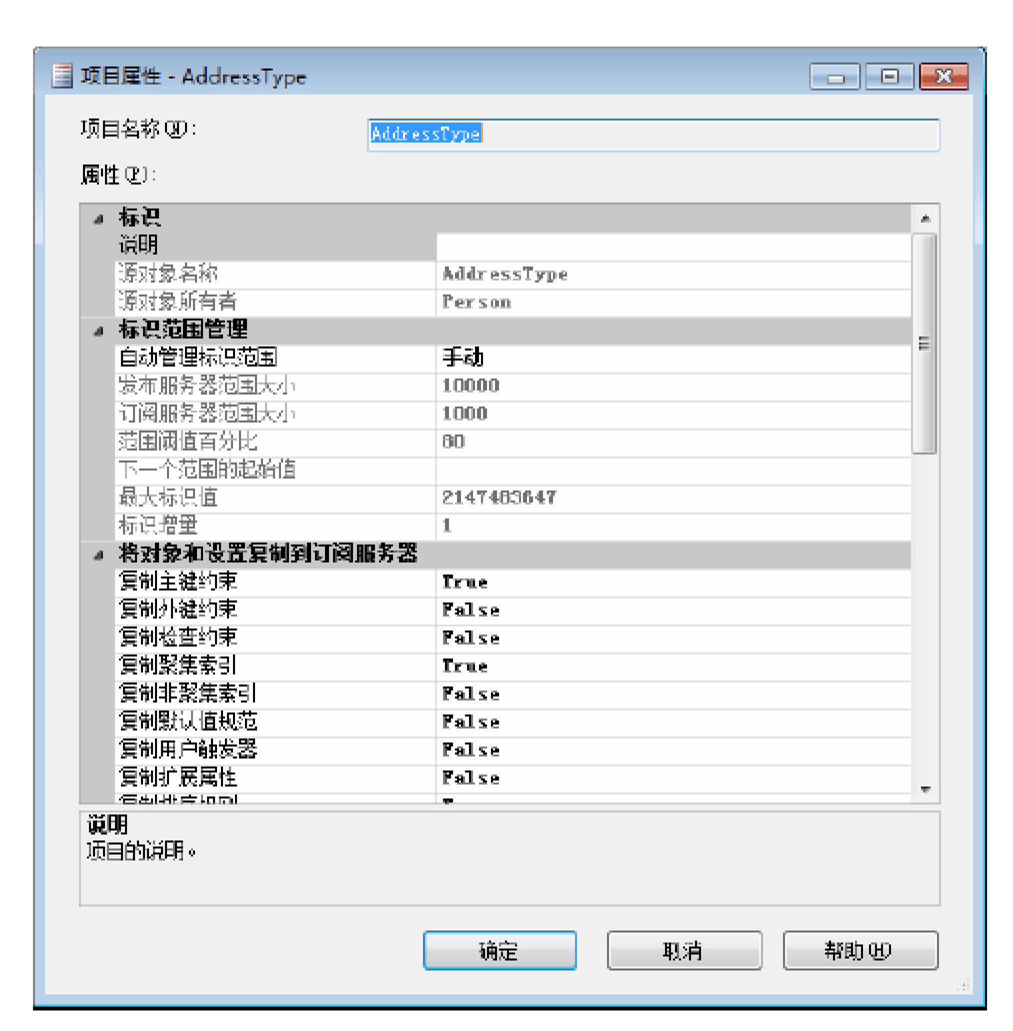


图 7.15　设置项目属性



图 7.16　“筛选表行”对话框

（11）假设这里需要筛选 Address 表，所有 City 为 Bothell 的都不进行发布，单击“添加”按钮，系统弹出“添加筛选器”对话框，如图 7.17 所示。

（12）选择要筛选的表为 Address 表，然后输入筛选语句：

```
SELECT <published_columns> FROM [Person].[Address] WHERE [City] !='Bothell'
```

单击“确定”按钮，回到筛选表行界面，同时系统将把筛选表和条件添加到“新建发布向导”对话框中，如图 7.18 所示。

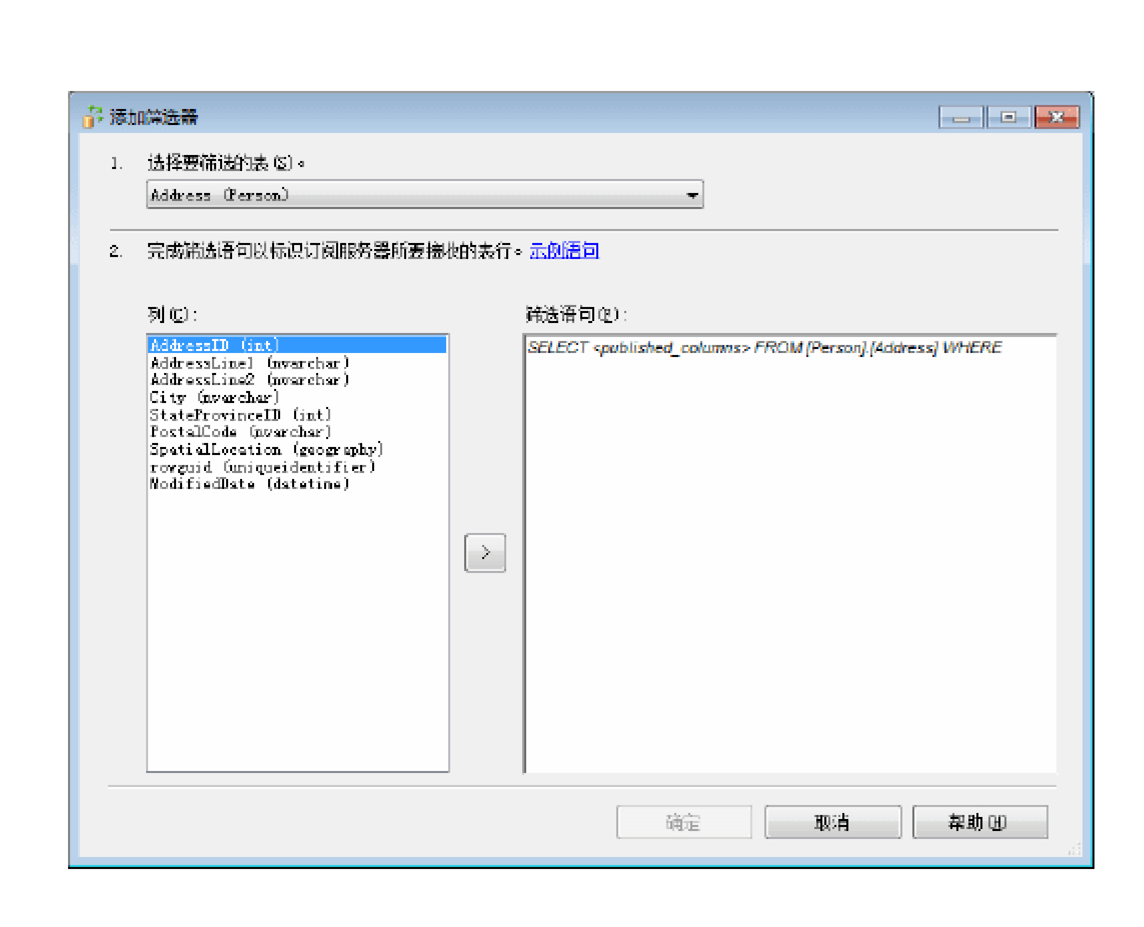


图 7.17　“添加筛选器”对话框

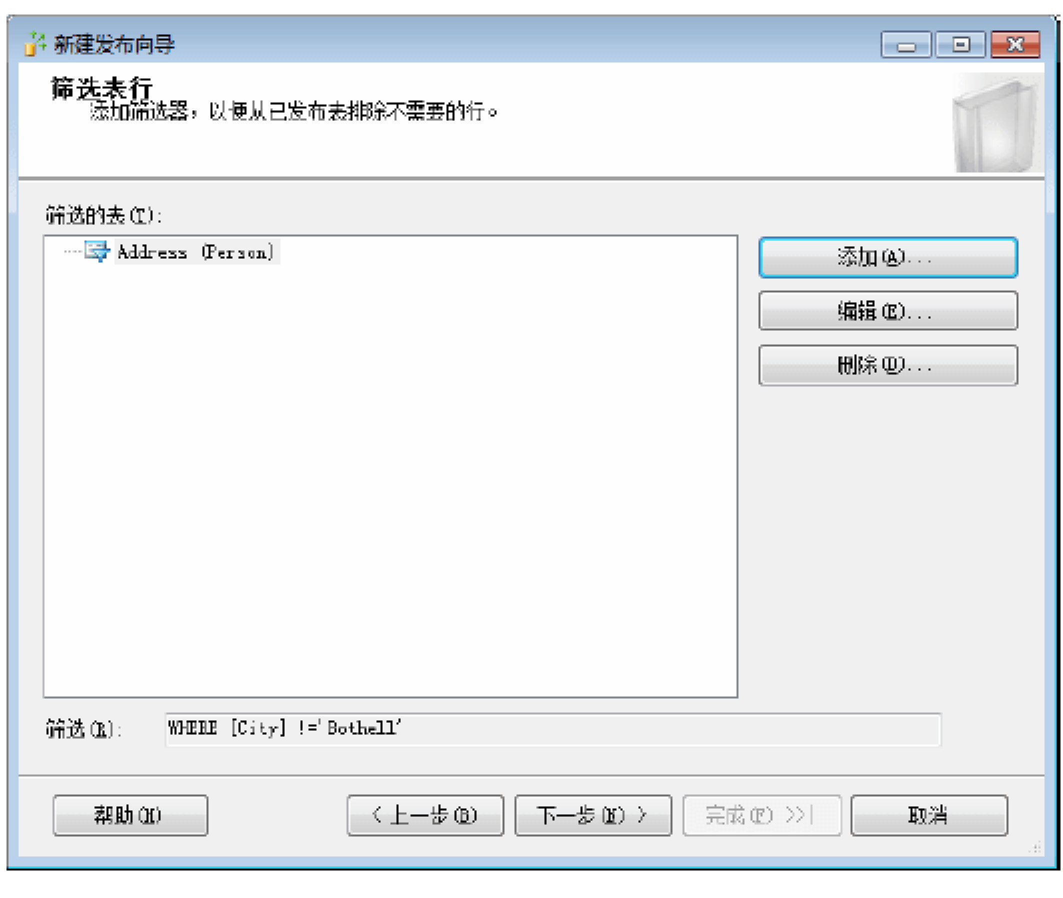


图 7.18　添加筛选表行

（13）单击“下一步”按钮，系统进入“快照代理”对话框，如图 7.19 所示。

该界面用于设置初始化订阅和快照代理执行的时间计划。选中“立即创建快照并使快照保持可用状态，以初始化订阅”复选框和“计划在以下时间运行快照代理”复选框，由于希望本次示例能尽快看到复制的效果，所以单击“更改”按钮将快照代理设置为每 3 分钟运行一次。

说明： 在实际系统中对于不需要频繁复制的数据可以设置为每天晚上 1：00 或其他业务空闲期运行一次，对于需要频繁复制的系统也不可将快照代理的执行频率设置过高；否则快照代理将影响生产环境系统性能。

（14）设置完快照代理，单击“下一步”按钮，进入“代理安全性”对话框，如图 7.20 所示。

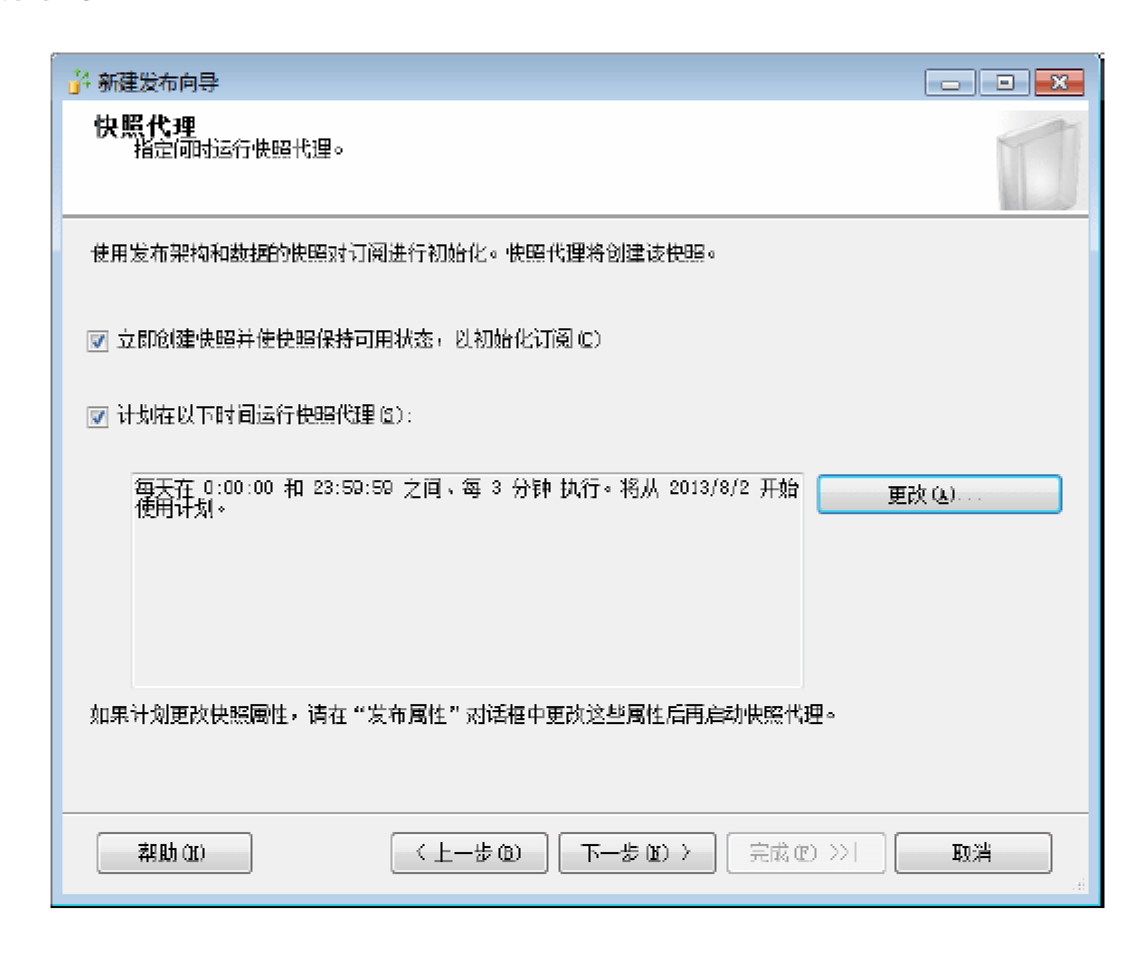


图 7.19　“快照代理”对话框

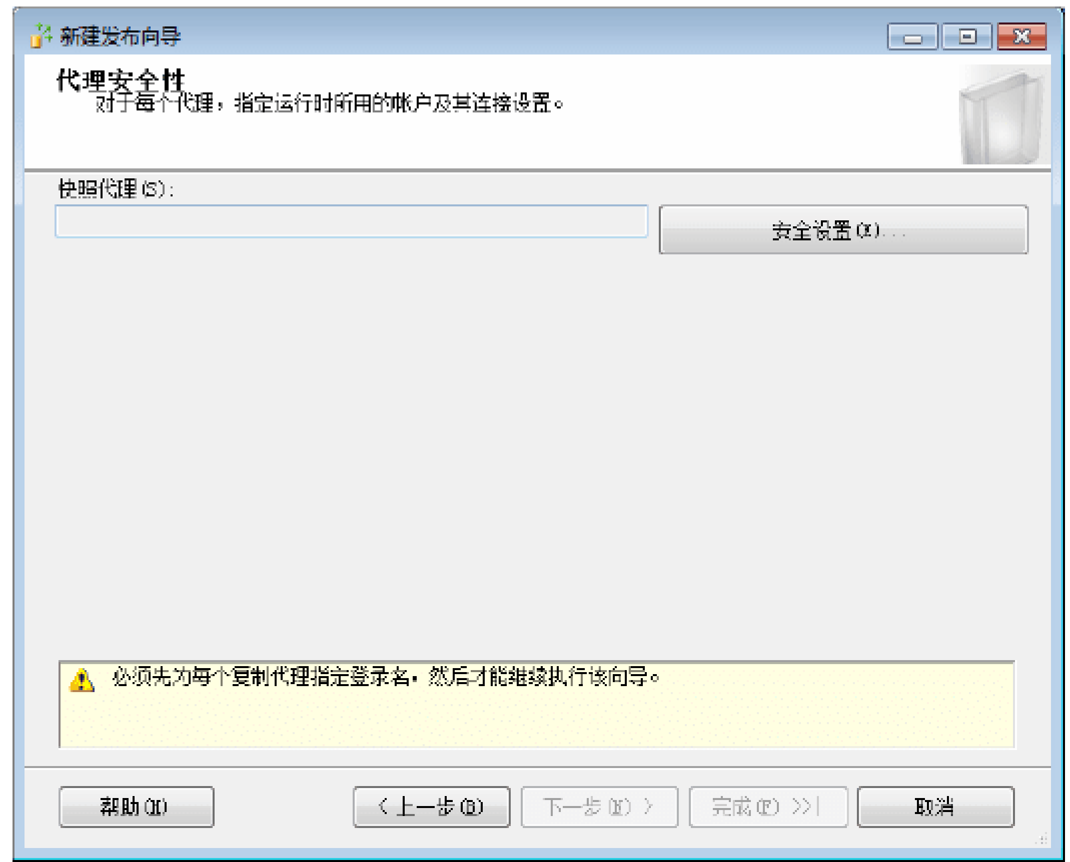


图 7.20　“代理安全性”对话框

（15）单击“安全设置”按钮，系统弹出“快照代理安全性”对话框，如图 7.21 所示。这里需要确认当前服务器中代理服务所运行的账户角色，如果代理用户角色在 Windows 中权限太低不能运行快照代理生成快照文件，则需要指定其他 Windows 账户。由于发布服务器和分发服务器是同一台，所以将通过模拟进程账户连接到发布服务器。单击“确定”按钮回到发布向导页面。

（16）单击“下一步”按钮，系统进入“向导操作”对话框，用于指定向导结束时进行的操作，如图 7.22 所示。

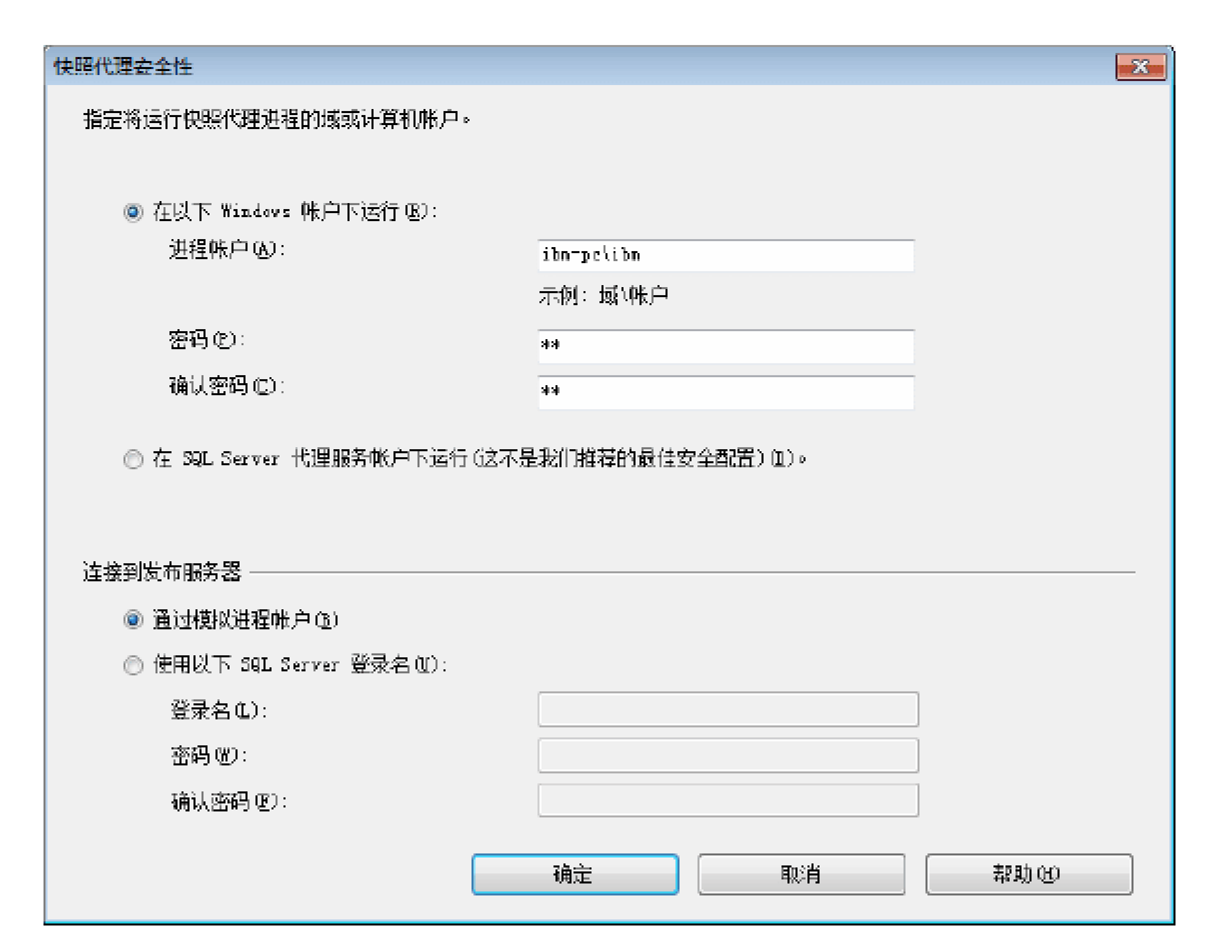


图 7.21　“快照代理安全性”对话框



图 7.22　“向导操作”对话框

选中在向导结束时“创建发布”复选框，如果需要生成创建发布的 SQL 脚本可以选中“生成包含创建发布的步骤的脚本文件”复选框。

（17）单击“下一步”按钮，进入向导的最后一步，为发布命名并完成向导界面，如图 7.23 所示。在“发布名称”文本框中输入名称比如 TestPublish，单击“完成”按钮，系统开始按照要求配置发布和分发并启用快照代理。运行完成后系统将给出成功提示，如图 7.24 所示。

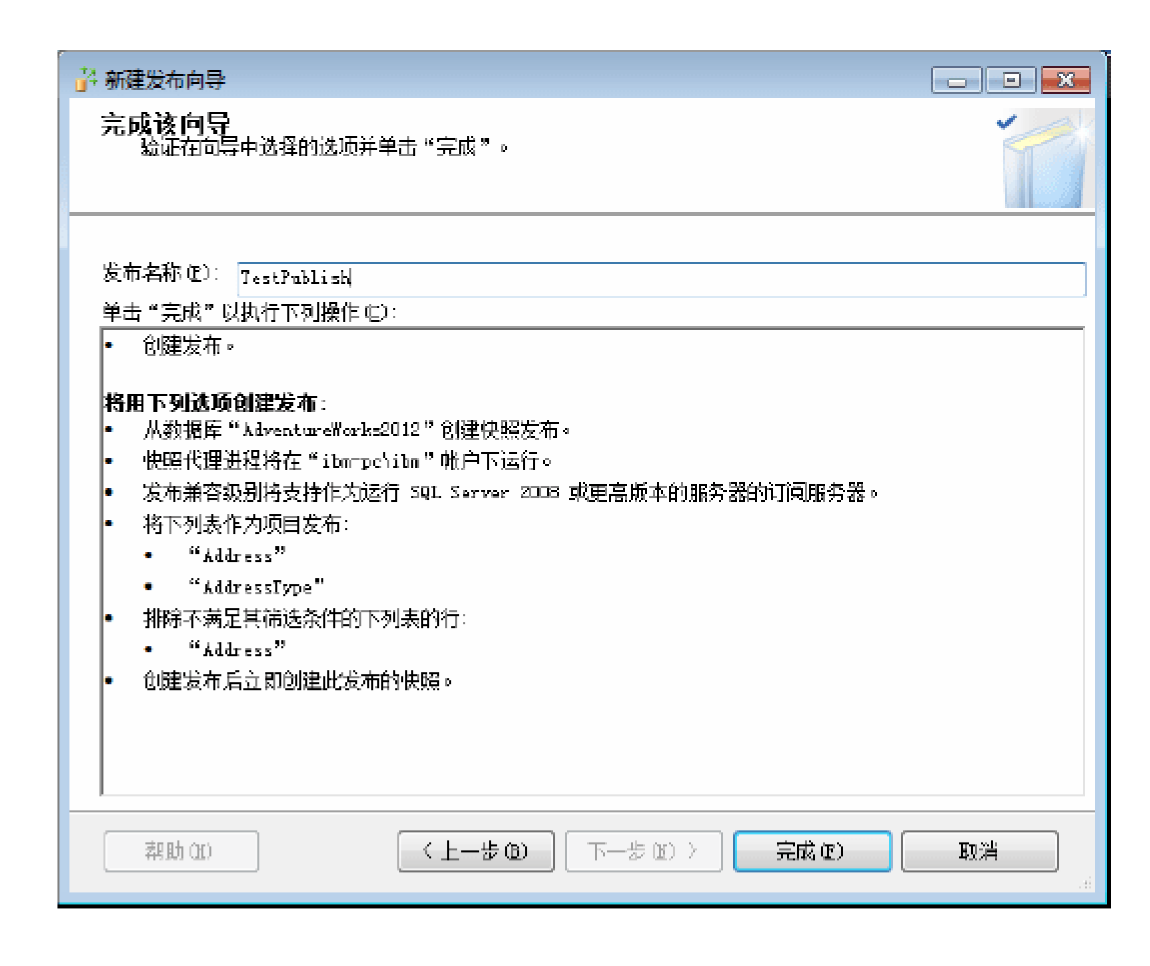


图 7.23　完成向导

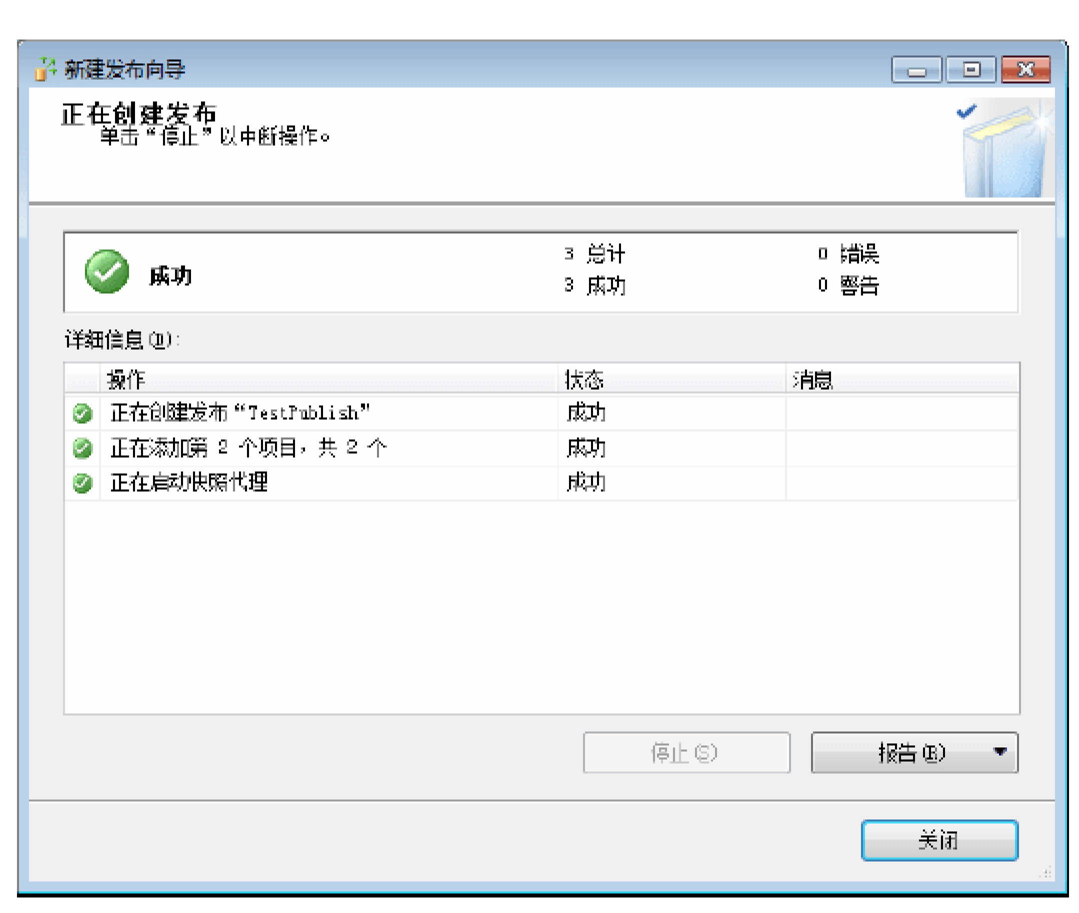


图 7.24　创建发布成功提示

根据配置快照文件已经生成，快照代理也已启用，系统会每隔 3 分钟在共享文件夹 C:\Share 中创建一个当前时间的快照文件夹，并将相关数据库对象的快照建立在文件夹中。另外，通过 SSMS 也可以在对象资源管理器中看到“本地发布”节点下有刚新建的快照发布，以及 SQL 代理下也多了几个用于快照复制的作业，如图 7.25 所示。

复制
本地发布
[AdventureWorks2012]: TestPublish
本地订阅
AlwaysOn 高可用性
管理
Integration Services 目录
SQL Server 代理
作业
distribution 的复制监视刷新器
IBM-PC-AdventureWorks2012-TestPublish
syspolicy_purge_history
代理历史记录清除: distribution
分发清除: distribution
复制代理检查
过期的订阅清除
重新初始化未通过数据验证的订阅

图 7.25　快照发布生成的作业

7.4.3　配置快照订阅

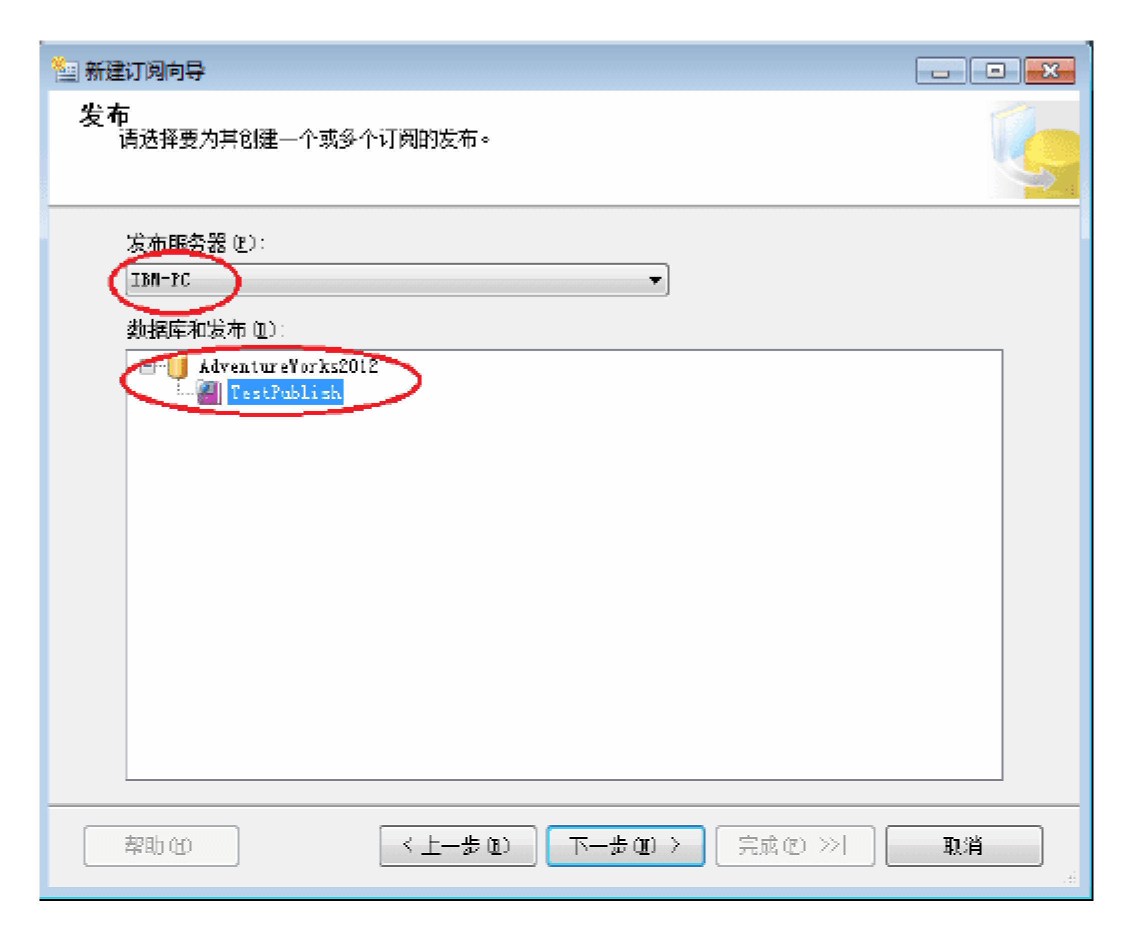
配置快照复制的订阅是在另外一台订阅服务器上进行。同快照发布类似，快照订阅也是在向导下完成，具体操作如下所述。

（1）在订阅服务器上打开 SSMS 并连接到本机。

（2）在对象资源管理器中展开“复制”节点下的“本地订阅”节点，右击“本地订阅”节点，在弹出的快捷菜单中选择“新建订阅”选项，系统将弹出“新建订阅向导”对话框。

（3）单击“下一步”按钮，系统进入“发布”对话框，如图 7.26 所示。在“发布服务器”下拉列表框中选择前面新建的发布服务器 IBM-PC，系统会将该服务器上配置的所有发布都显示出来，这里只配置了 TestPublish，所以只显示该发布。

（4）单击“下一步”按钮，系统进入“分发代理位置”对话框，在数据库复制中有推送订阅和请求订阅两种，这里就使用默认值请求订阅，如图 7.27 所示。



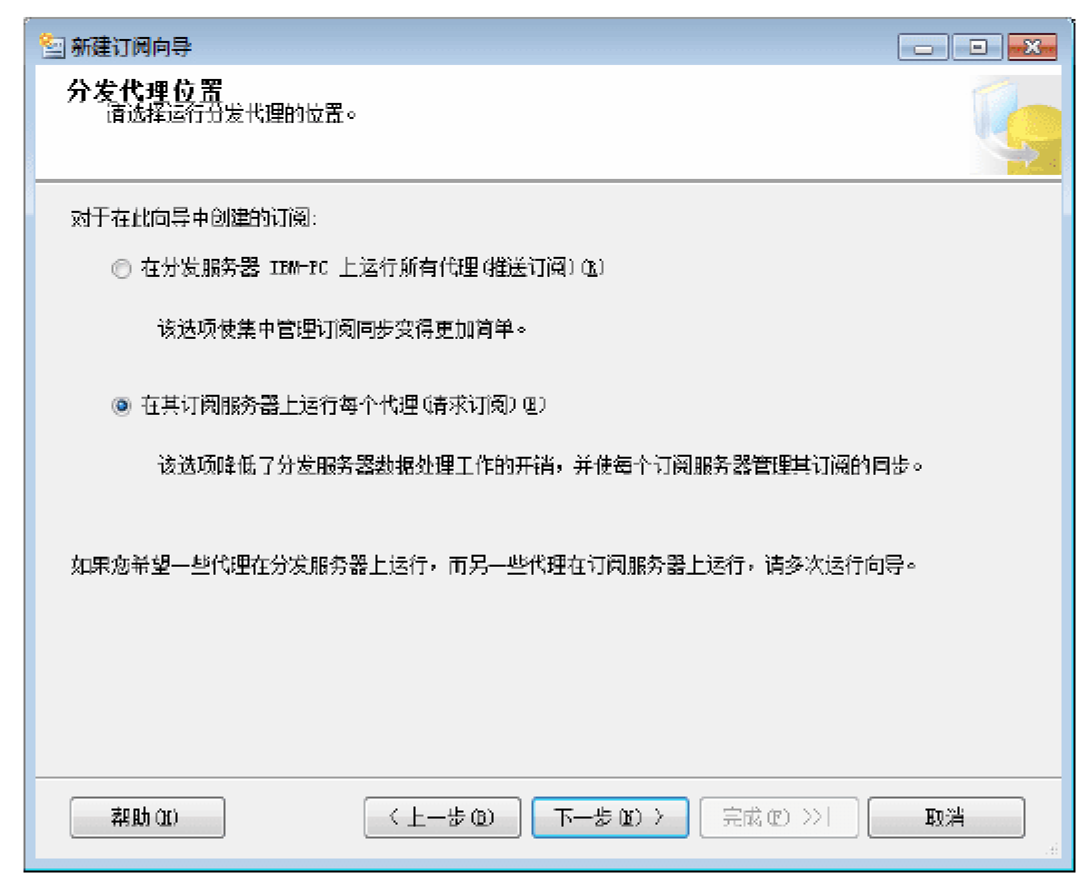




图 7.26　“发布”对话框　　　　图 7.27　“分发代理位置”对话框

（5）单击“下一步”按钮，进入“订阅服务器”对话框，如图 7.28 所示。这里选择“订阅数据库”下拉列表框中的“新建数据库”命令，系统将弹出新建数据库窗口。这里我们创建新的数据库 AdventureWorks2 用于订阅数据库。

（6）单击“下一步”按钮，系统进入“分发代理安全性”对话框，如图 7.29 所示。

（7）在其中单击“…”按钮，系统弹出“分发代理安全性”对话框，如图 7.30 所示。其中选择推荐的“在 SQL Server 代理服务账户下运行（这不是我们推荐的最佳安全配置）”单选按钮，通过模拟进程账户连接到分发服务器，同样通过模拟进程账户连接到订阅服务

器。选定后单击“确定”按钮回到向导主界面。

（8）单击“下一步”按钮，系统进入“同步计划”对话框，如图 7.31 所示。

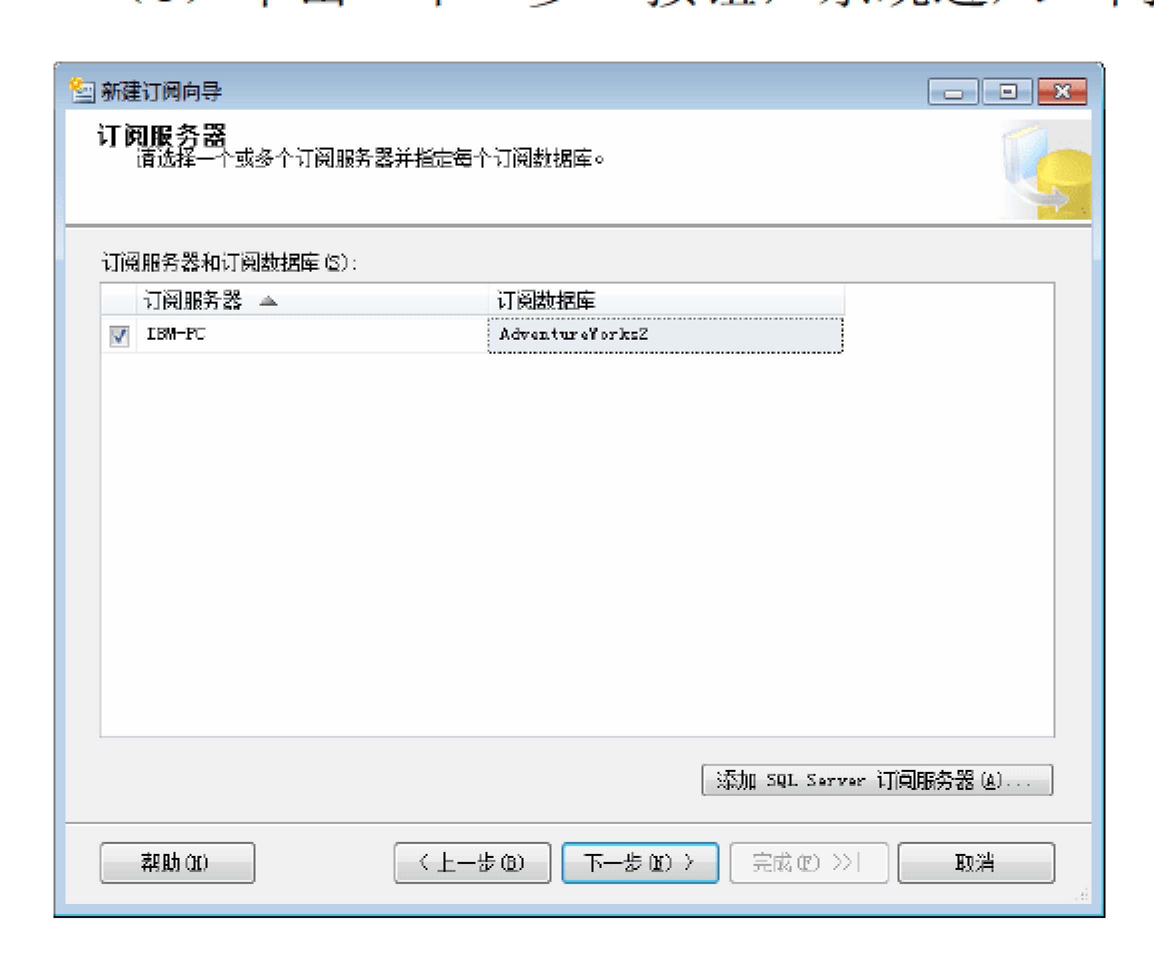


图 7.28　“订阅服务器”对话框

图 7.29　“分发代理安全性”对话框

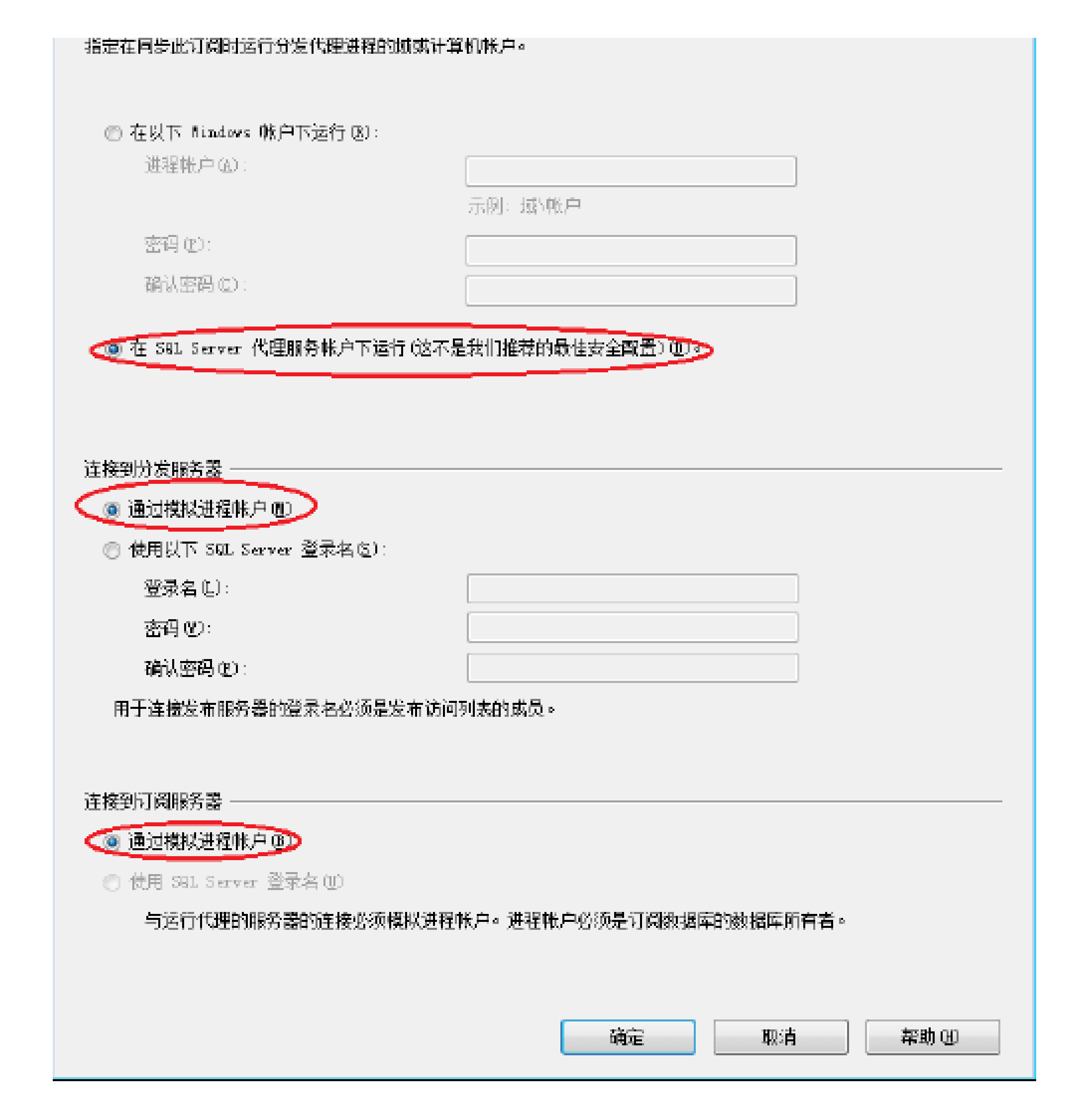


图 7.30　分发代理安全性具体配置



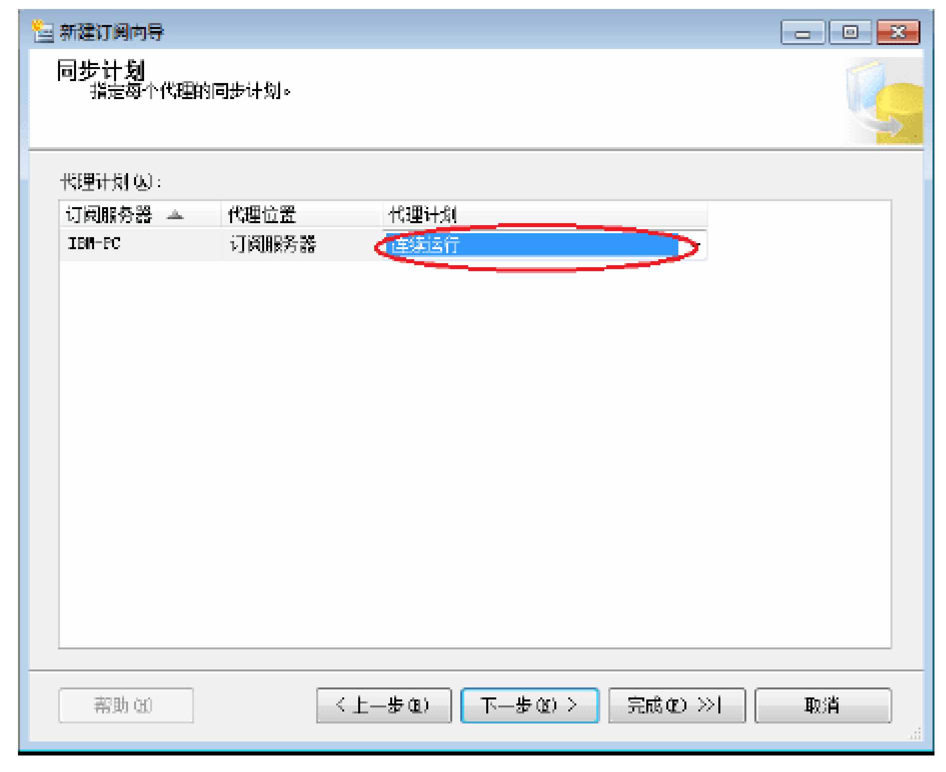


图 7.31　“同步计划”对话框

这里可以选择连续运行、仅按需运行或者按照自定义的计划运行 3 种运行方式。连续运行是实时监控发布服务器，当发布服务器有了新的发布时就自动更新订阅。按需运行是全手动的运行方式，在需要同步的时候手动启动订阅代理。按自定义计划运行就是按照计划定义的时间和频率（比如每天晚上 1：00）自动执行订阅代理。这里就使用连续运行代理计划。

（9）单击“下一步”按钮，系统进入“初始化订阅”对话框，如图 7.32 所示。“初始化时间”下拉列表框中有立即和首次同步时两种选择，如果不希望初始化订阅服务器的话可以取消对“初始化”复选框的选择。这里选择立即初始化。

（10）单击“下一步”按钮，进入订阅配置的最后一步配置界面，配置向导结束时执行的操作，如图 7.33 所示。

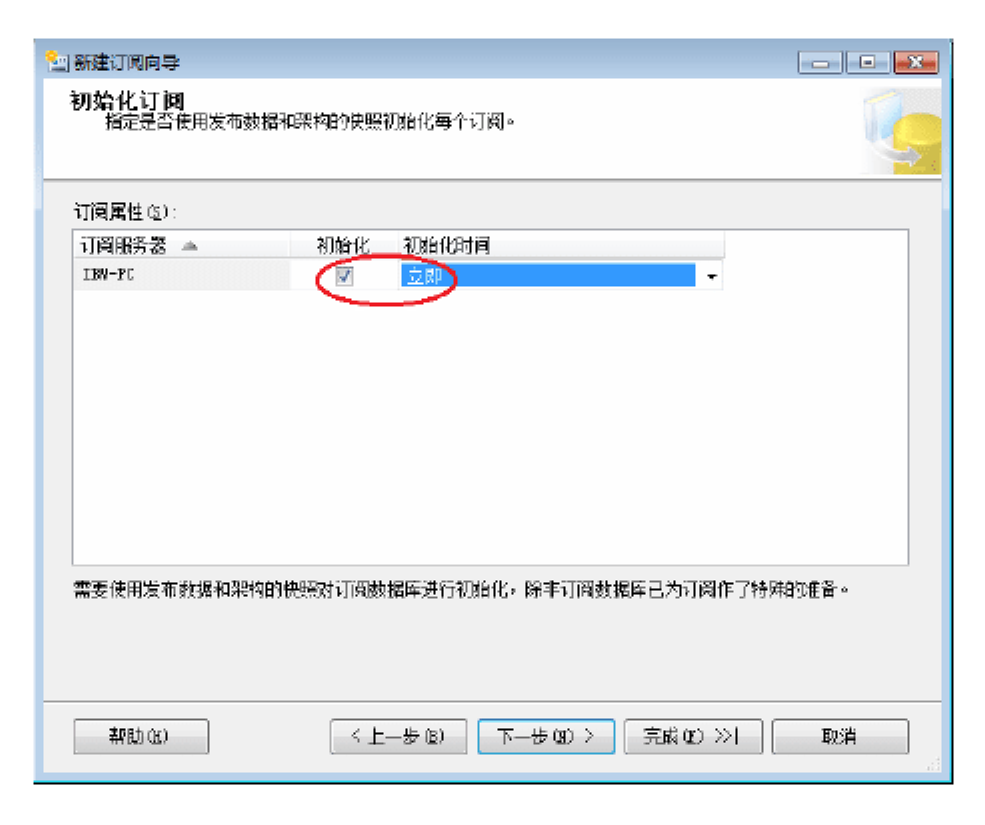


图 7.32　“初始化订阅”对话框

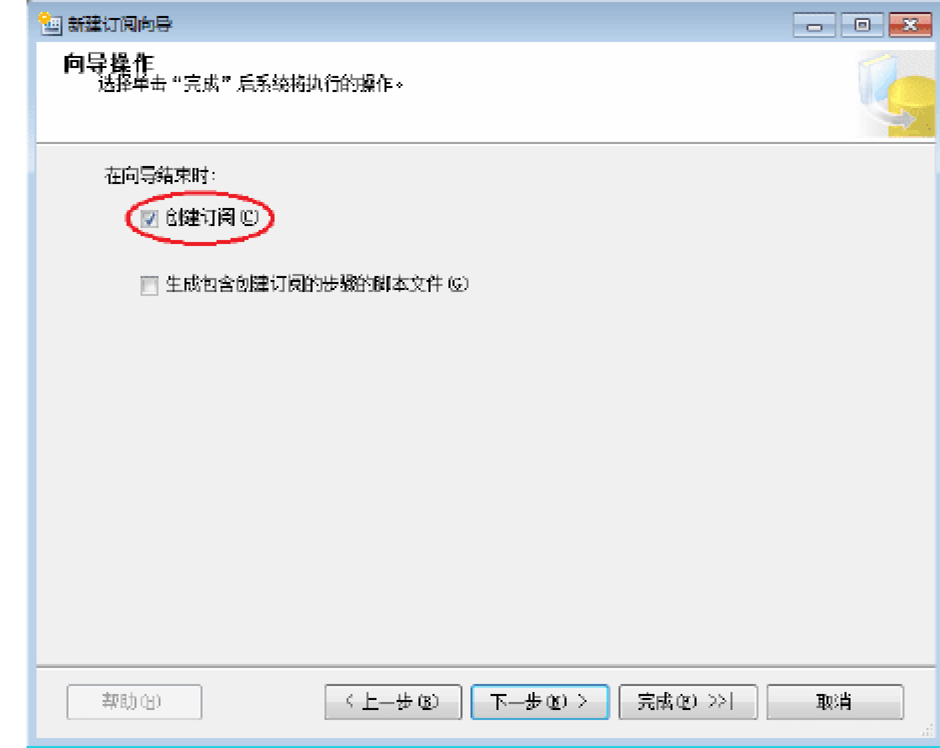


图 7.33　“向导操作”对话框

若要在向导结束时创建订阅，则选中“创建订阅”复选框；若要在向导结束时生成包含创建订阅的步骤的 T-SQL 脚本，则选中“生成包含创建订阅的步骤的脚本文件”复选框。这里只选中“创建订阅”复选框。

（11）单击“下一步”按钮，再单击“完成”按钮完成订阅的配置。过几秒钟后系统就完成了订阅的配置，如图 7.34 所示。

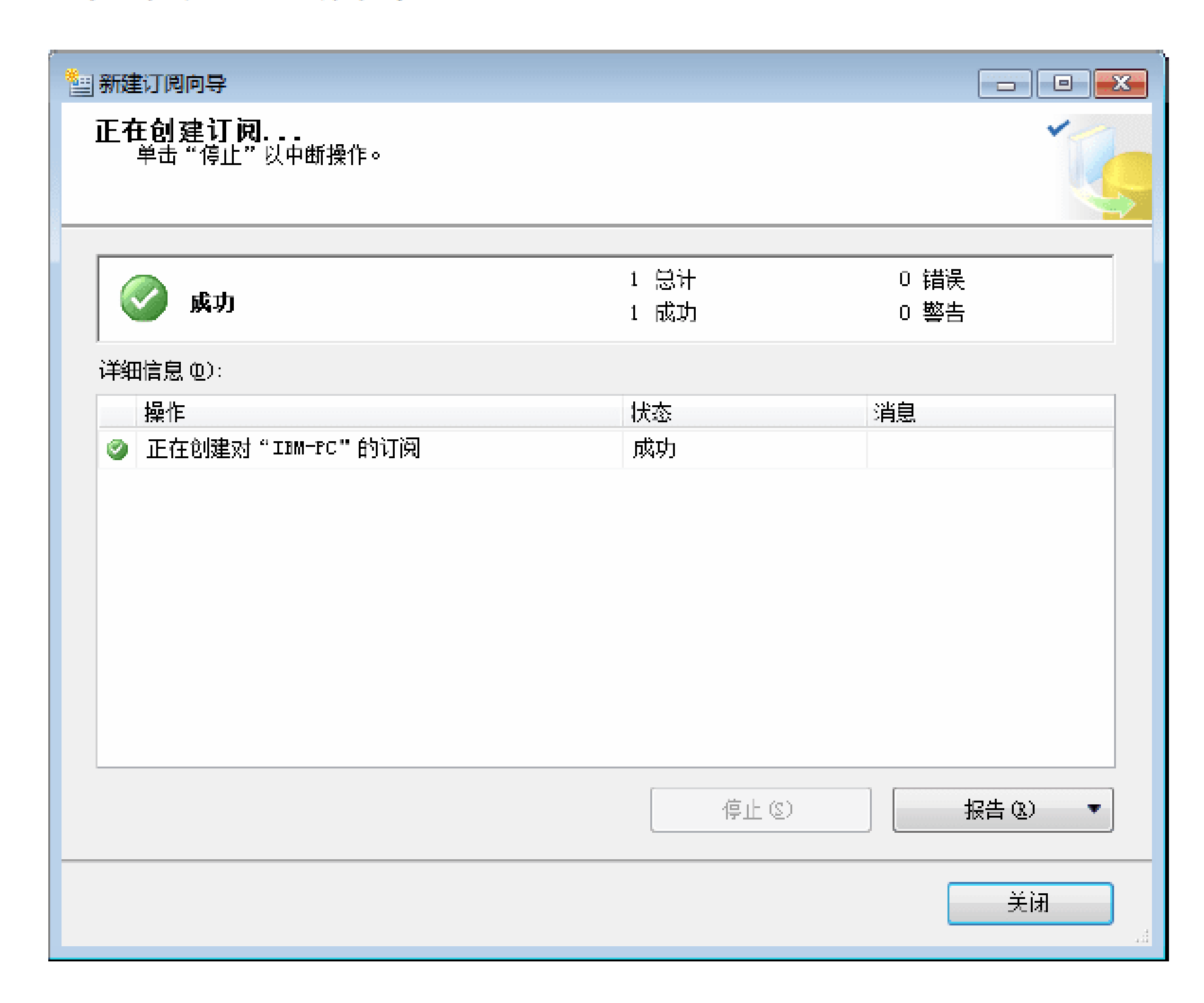


图 7.34　创建订阅成功提示

至此快照复制已经配置完成，打开 AdventureWorks2 数据库可以看到发布的 Address 表和 AddressType 表。

7.4.4　配置事务复制和合并复制

事务复制的配置过程与快照复制并没有太大的不同，同样是通过发布向导来完成。主要是在选择发布类型时选择“事务发布”选项或者“具有可更新订阅的事务发布”选项，如图 7.35 所示。一般的事务发布和具有可更新订阅的事务发布的主要区别如下：

- 事务发布在订阅服务器收到已发布数据的初始快照后，发布服务器将事务流式传输到订阅服务器。

- ❑ 具有可更新订阅的事务发布在 SQL Server 订阅服务器收到已发布数据的初始快照后，发布服务器将事务流式传输到订阅服务器。来自订阅服务器的事务被应用于发布服务器。

简单地说，事务发布只是单方面的将事务日志从发布服务器传输到订阅服务器，而事务复制的可更新订阅允许订阅服务器将更改复制到发布服务器。

另外一点与快照发布不同的是，事务发布需要配置日志读取代理的安全设置，而可更新订阅的事务发布不但要配置日志读取代理的安全设置也还要配置队列读取器代理的安全设置。

合并复制的配置在图 7.35 中选择合并发布后单击“下一步”按钮，进入“订阅服务器类型”对话框，此时要选择订阅服务器类型。合并复制支持 SQL Server 2012、SQL Server 2008、SQL Server 2005 和 SQL Server 2000 类型的订阅服务器，如图 7.36 所示。

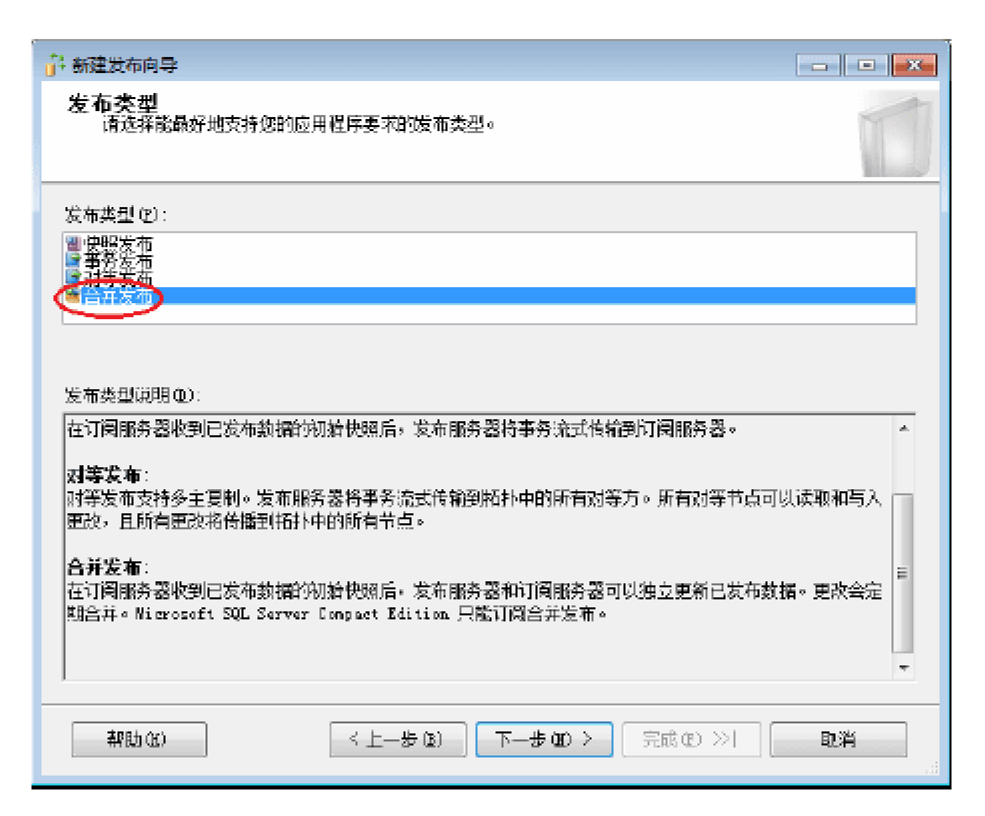


图 7.35　“发布类型”对话框

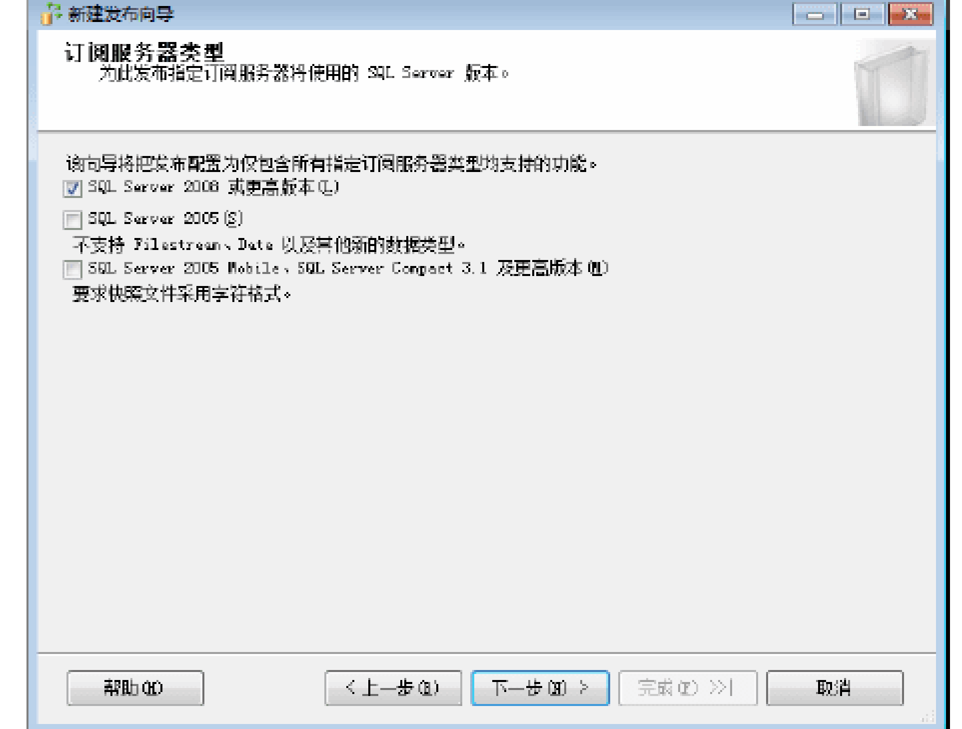


图 7.36　“订阅服务器类型”对话框

事务复制的订阅以及合并复制的订阅与快照复制的订阅相似，这里不再介绍。

7.5　管 理 复 制

复制在创建后需要进行监视、优化、修改和删除等维护操作，以满足不断变化的情况和提高系统的性能及可用性。本节就从管理复制的角度来介绍数据库复制的维护操作。

7.5.1　添加项目

可以通过 SSMS 可视化操作、T-SQL 脚本和 RMO 编程来对发布进行维护。由于使用 T-SQL 脚本和 RMO 编程维护相对复杂，而且通过 SSMS 的可视化操作也能自动生成 T-SQL 脚本，所以这里只介绍通过 SSMS 操作的方式来维护发布。

创建发布后，可以添加和删除项目。可以随时添加项目，但删除项目所需的操作取决于复制的类型和删除项目的时间。

添加项目涉及的操作有：将项目添加到发布、为发布创建新的快照、同步订阅以应用新项目的架构和数据。以前面创建的 TestPublish 快照发布为例，该发布中只发布了两个表：

Address 和 AddressType，若要将另外一个表 Contract 添加到该发布中，则具体操作如下所述。

（1）使用 SSMS 连接到发布服务器，在对象资源管理器中展开“复制”节点下的“本地发布”节点。右击“[AdventureWorks2012]:TestPublish”节点，在弹出的快捷菜单中选择“属性”选项，系统将弹出“发布属性”对话框，如图 7.37 所示。

（2）选择“选择项”的“项目”选项，切换到发布项目界面。同时列出了当前发布中所包含的数据库对象，如图 7.38 所示。

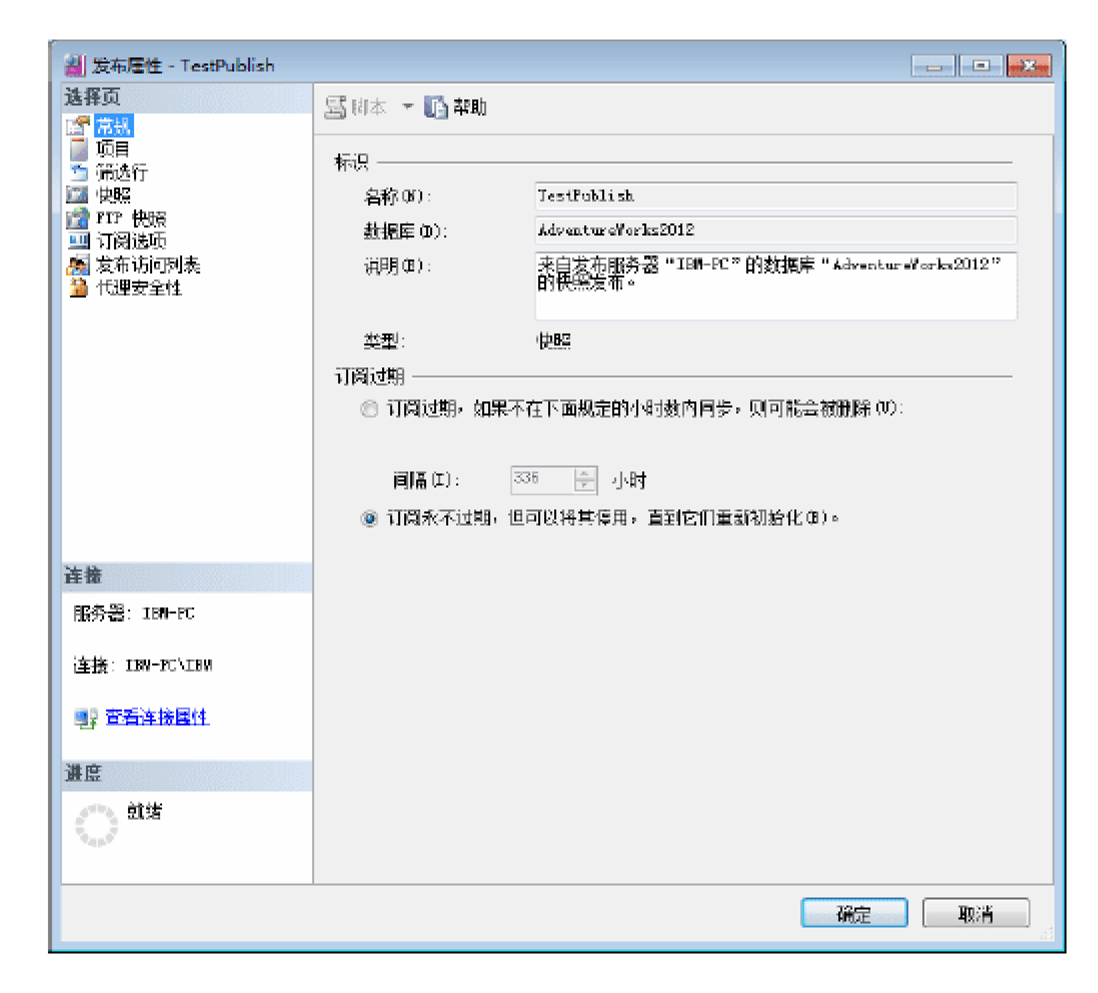




图 7.37 “发布属性”对话框

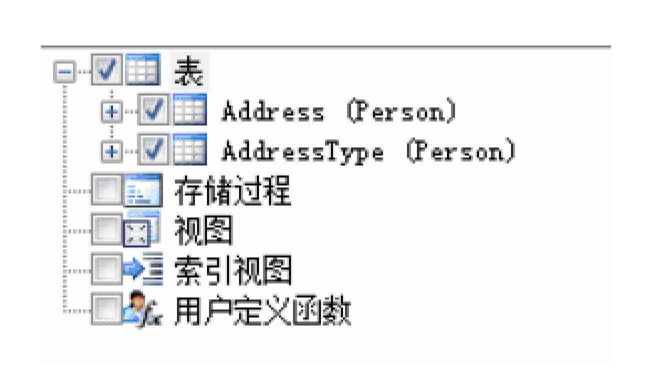


图 7.38 发布项目

（3）选择“表”复选框，系统将选中所有的表，然后再选择“表”复选框以取消对所有表的选中。此时系统提示“删除指定的项目将使当前的快照失效。这不会影响现有订阅。是否确实要删除所有表项目？”，单击“是”按钮取消了对所有表的选中。

（4）重新选中 Address、AddressType 和 ContractType 表左边的复选框。

（5）单击“确定”按钮，完成对 ContractType 表发布的添加工作。

（6）接下来是为发布创建新的快照。在对象资源管理器中右击“[AdventureWorks2012]:TestPublish”节点，在弹出的快捷菜单中选择“查看快照代理状态”选项，系统将弹出“查看快照代理状态”对话框，如图 7.39 所示。从图中可以看到，当前同步了 2 个项目的快照，而且上次同步时间为 6 秒钟。显然 ContractType 表还没有进行同步。

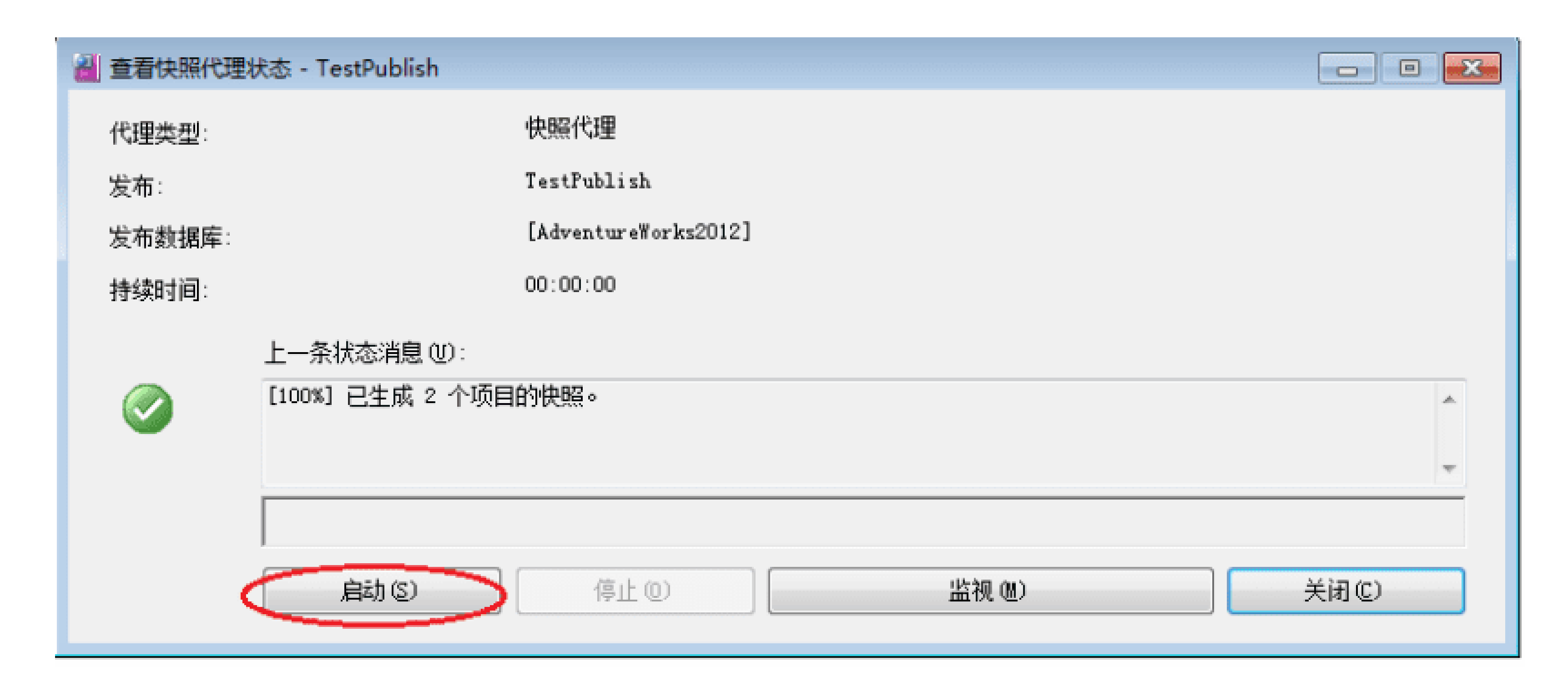


图 7.39 “查看快照代理状态”对话框

（7）单击“启动”按钮，系统将重新生成快照，将 Contract 表添加到快照中。

（8）连接到订阅服务器，在订阅项目上右击，在弹出的快捷菜单中选择“查看同步状

态”选项，系统将弹出“查看同步状态”对话框。该对话框与“查看快照代理状态”对话框相同。

（9）单击“启动”按钮，系统将重新同步订阅。

7.5.2　删除项目

可以随时从发布中删除项目，但必须考虑以下行为：

- 从发布中删除项目并不会将对象从发布数据库中删除，也不会将相应对象从订阅数据库中删除。必要时，使用 DROP<对象>删除这些对象。在删除通过外键约束与其他已发布项目相关的项目时，建议手动或使用按需脚本执行在订阅服务器上删除表，指定包含相应 DROP<对象>语句的脚本。
- 对于兼容级别为 90RTM 或更高的合并发布，可以随时删除项目，但需要创建一个新的快照。
- 如果项目在连接筛选器或逻辑记录关系中是父项目，就必须先删除关系，这需要重新初始化。
- 如果项目具有发布中的最后一个参数化筛选器，就必须重新初始化订阅。
- 对于兼容级别低于 90RTM 的合并发布，可以在初始同步订阅之前删除项目，而无需特别考虑。如果在同步一个或多个订阅之后删除项目，则必须删除、重新创建和同步订阅。
- 对于快照发布或事务性发布，可以在创建订阅之前删除项目，而无需特别考虑。如果在创建一个或多个订阅之后删除项目，则必须删除、重新创建和同步订阅。使用 sp_dropsubscription 可以删除订阅中的单个项目，而不是删除整个订阅。

删除项目的操作与添加项目的操作类似，只是在发布属性窗口的“项目”选项卡下，将需要删除项目的复选框设置为不选中，然后单击“确定”按钮即可。

7.5.3　复制监视器

Microsoft SQL Server 复制监视器是一个图形工具，可用来监视复制拓扑的整体运行状况。复制监视器提供一个以发布服务器为主的视图，以两个窗格的形式显示所有复制活动。在监视器的左窗格中添加发布服务器后，监视器的右窗格中即显示发布服务器、其发布、对这些发布的订阅和各种复制代理的相关信息。除了显示有关复制拓扑的信息以外，复制监视器还可用于执行多种任务，如启动和停止代理以及验证数据。

在 SSMS 中右击“复制”节点，在弹出的快捷菜单中选择“启用复制监视器”选项便可启动复制监视器。也可以输入命令行 sqlmonitor 来启动复制监视器。

“复制监视器”对话框的左边显示发布服务器组、发布服务器和发布。 若要在复制监视器中查看任何信息，必须首先添加发布服务器。

复制监视器在下面的 3 选项卡上显示发布服务器的相关信息：

- “发布”选项卡提供发布服务器上所有发布的摘要信息。
- “订阅监视列表”选项卡用于显示所选发布服务器上所有可用发布的订阅的信息。可以筛选订阅列表来查看有错误的订阅、有警告的订阅以及任何执行不当的订阅。

使用该选项卡可以访问订阅属性、访问与订阅关联的代理的详细信息、重新初始化订阅以及验证订阅。

说明：对于运行 Microsoft SQL Server 2005 以前版本的分发服务器，不显示订阅监视列表选项卡。

- “代理”选项卡显示所有类型的复制所用代理和作业的详细信息。使用该选项卡，还可以启动和停止每个代理和作业。

7.5.4　提高复制性能

在数据库复制中影响数据库性能的因素有服务器和网络硬件、数据库设计、分发服务器配置、发布设计和选项、筛选器设计和用法、订阅选项、快照选项、代理参数、维护。配置复制后，建议制定一个性能标准，以便于确定复制在通常用于应用程序和拓扑的常见工作负荷中的行为方式。使用复制监视器和系统监视器确定复制性能的以下 5 个维度的典型数目。

- 滞后时间：在复制拓扑中的节点之间传播数据更改所用的时间。
- 吞吐量：系统在某段时间内可以承受的复制活动量（通过某个时间段内传递的命令数量来度量）。
- 并发：可以在系统上同时运行的复制进程数。
- 同步持续时间：完成给定同步所用的时间。
- 资源消耗：用做复制处理结果的硬件和网络资源。

滞后时间和吞吐量与事务复制关系最密切，因为建立在事务复制基础上的系统通常需要低滞后时间和大吞吐量。并发和同步持续时间与合并复制关系最密切，因为建立在合并复制基础上的系统通常具有大量的订阅服务器，并且发布服务器可能具有大量与这些订阅服务器的并发同步。确定基准数目后，应在复制监视器中设置阈值。可以通过在复制监视器中设置阈值和警告也可以为复制代理事件使用警报来监视复制性能。

关于服务器和网络以及数据库设计对数据库性能的影响这里就不多说了。这里主要讲解如何设置发布、订阅和快照以提高复制性能。在发布的设计中应注意以下几点：

- 仅发布所需的数据。
- 通过发布设计和应用程序行为最大限度地减少冲突。
- 灵活地使用行筛选器。

订阅服务器的设置要注意：

- 当存在大量订阅服务器时，应使用请求订阅。
- 如果订阅服务器滞后太多，则应考虑重新初始化订阅。

快照设置的注意事项有：

- 仅在必要时和非高峰时段运行快照代理。
- 除非要求使用字符模式快照，否则使用本机模式快照。
- 为一个发布使用一个快照文件夹。
- 将快照文件夹放在分发服务器上未用于存储数据库或日志文件的本地驱动器中。
- 在订阅服务器上创建订阅数据库时，考虑指定简单恢复模式或大容量日志恢复

模式。

- 对于低带宽网络，考虑在可移动媒体上使用备用快照文件夹和压缩的快照。
- 考虑手动初始化订阅。

从以上方面出发，将有助于提高数据库复制的性能，但是具体设置还是要根据实际情况做一定的调整，在满足业务需求的情况下提高性能。

7.6　小　　结

本章详细介绍了 SQL Server 复制的基本概念和配置方法，并讨论了不同类型复制的工作原理和复制的管理。

本章以大批量数据的导入导出为核心，首先从 bcp 工具开始介绍，再发散到 BULK INSERT 命令和 OPENROWSET 命令，这些工具和命令都可以实现同类数据库或异类数据库之间的批量数据传输。接下来以 SQL Server 复制为中心，介绍了复制的各种基本概念和各种类型复制的工作机制，从一个数据库快照发布和订阅的配置示例讲解了数据库复制的配置过程。最后从管理的角度讲解了发布的维护、复制的监视和数据库性能的提高。

本章关于复制的配置操作都是在 SSMS 下进行的，而没有采取 SSMS 配置和 T-SQL 配置双讲解的方式，主要是因为关于数据库复制的 T-SQL 命令较多而且复杂，对于大多数人来说并不需要了解每个命令及每个参数的含义。同时，SSMS 下的发布和订阅向导为用户提供了将配置导出为 T-SQL 脚本的功能，所以若是要用 T-SQL 进行配置，只需要在向导中设置将配置导出为 T-SQL 脚本即可。

第 3 篇　SQL Server 开发

第 8 章　数据库设计

在前面章节已经了解了数据的一些基础知识，诸如表、联接、约束、视图、存储过程和函数等都是设计和管理 SQL Server 2012 数据库的基础。本章将从整个关系数据库概念出发，讲解数据库的设计。

8.1　实体——关系模型

实体——关系模型又被简称为 E-R 模型，是关系数据库设计中最基本的概念。实体—关系模型的提出有助于数据库的设计。E-R 模型是一种语义模型，模型的语义方面主要体现在模型力图表达数据的意义。E-R 模型在将现实世界的含义和相互关联映射到概念模式方面非常有用。只有了解了什么是实体，什么是实体的关系后才能明白为什么表要这样建，为什么这两个表要直接建立外键。

8.1.1　基本概念

关系数据模型把概念模型中实体，以及实体之间的各种联系均用关系来表示。从用户的观点来看，关系模型中数据的逻辑结构是一张二维表，它由行列构成。如图 8.1 所示为一个员工人事数据表。

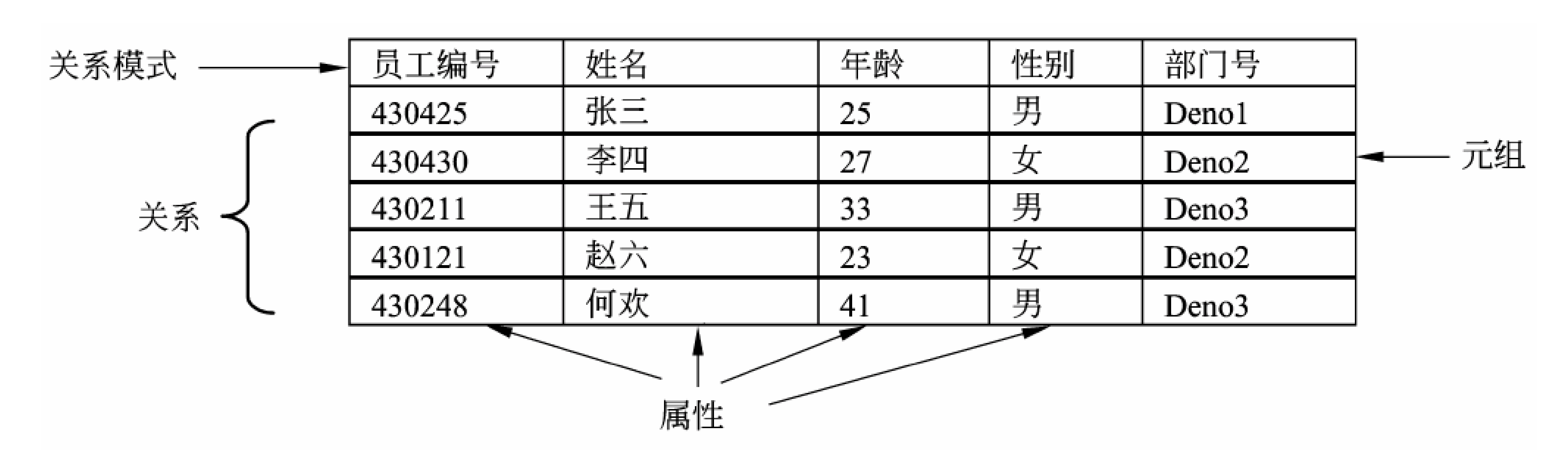


图 8.1　基本术语

下面就以此图为例，介绍一下关系数据模型中的几个基本概念。

（1）关系：每一个关系都可以用一张二维表来表示，常称为表。每一个关系表都有个区别于其他关系表的名字，称关系名。关系是概念模型中同一类实体，以及实体之间联系集合的数据模型表现。如图 8.1 中的员工人事数据表。

（2）属性：表中的每一列即称之为一个属性，每个属性都有一个显示在每一列首行的

属性名。在一个关系表当中属性名不能重复。如图 8.1 中有 5 列，对应 5 个属性（员工编号，姓名，年龄，性别，部门号）。关系的属性对应概念模型中实体型以及联系的属性。

（3）域：在关系中每个属性的值是有一定变化范围的，如图 8.1 员工人事数据表，其中属性“员工编号”的变化范围是 6 位字符；属性“姓名”的变化范围是 6 位字符；属性“年龄”的变化范围是 0～100 岁；属性“性别”的变化范围只能是男、女两个值。属性“部门号”的变化范围是所有可能的部门集合。

每一个属性所对应的变化范围叫做属性的变域或简称域，它是属性值的集合，关系中所有属性的实际取值必须来自于它对应的域。例如，属性“员工编号”的域是 6 位字符，因此“员工编号”中出现的所有取值的集合必须是该域上的一个子集。

（4）元组：二维表中的每一行数据称为一个元组或记录。一个元组对应概念模型中一个实体的所有属性值的总称。如图 8.1 所示有 5 行数据，也就有 5 个元组。由若干个元组就可构成一个具体的关系，一个关系中不允许有两个完全相同的元组。

（5）分量：一个元组在一个属性域上的取值称为该元组在此属性上的分量。

（6）关系模式：二维表的表头行称为关系模式，即一个关系的关系名及其全部属性名的集合。关系模式是概念模型中实体型以及实体型之间联系的数据模型表示。一般表示为：

```
关系名（属姓名 1，属性名 2，……，属性名 n）
```

图 8.1 中员工认识数据表中的关系模式为：

```
员工信息表（员工编号，姓名，年龄，性别，部门号）
```

关系模式和关系是型与值的联系。关系模式指出了一个关系的结构；而关系则是由满足关系模式结构的元组构成的集合。因此关系模式决定了关系的变化形式，只要关系模式确定了，由它所产生的值——关系也就确定了。关系模式是稳定的、静态的，而关系则是随时间变化的、动态的。但通常在不引起混淆的情况下，两者可都成为关系。

说明：一个具体的关系数据库是一个关系的集合，而关系数据库模式是关系模式的集合。

8.1.2 实体集

实体（Entity）是一个数据对象，指可以区别客观存在的事物，如人、部门、表格、项目等。同一类实体的所有实例的集合就构成该对象的实体集（entity set）。也就是说，实体集是实体的集合，由该集合中实体的结构形式表示，而实例则是实体集中的某一个特例，例如，销售订单 6380 号是销售实体集的一个实例，通过其属性值表示。实体使用矩形来表示，矩形中为实体的名称，如图 8.2 所示。

实体名称

图 8.2 实体表示

在 E-R 模型中，实体用方框表示，方框内注明实体的命名。实体名常用以大写字母开头的有具体意义的英文名词表示，联系名和属性名也采用这种方式。通常实体集中有多个实体实例。例如，数据库中存储的每笔订单都是销售实体集的实例。如表 8.1 所示为一个

实体集和它的两个实例。

表 8.1　实体集和两个实例

学 生 实 体	实　例　1	实　例　2
学号	2303002001	2303002002
姓名	张三	李四
出生日期	1981-1-9	1982-12-29
性别	男	女

实体集不一定互不相交。例如可以定义员工的实体集 Employee 和所有客户的实体集 Customer。而一个 Person 实体可以是 Employee 实体，也可以是 Customer 实体，可以既是 Employee 实体也是 Customer 实体，还可以都不是。

实体通过一组属性来表示。属性是实体集中每一个成员所拥有的描述性性质。为某实体集设计一个属性表明数据库为该实体集中每个实体存储相似的信息，但每个实体在每个属性上都有各自的值。例如 Customer 实体集可能具有属性 CustomerID、CustomerName 和 CustomerCity 及其他更多的信息。实体每个属性都有一个值。例如表 8.2 所示为 Customer 实体集的部分数据。

表 8.2　Customer实体集

CustomerID	CustomerName	CustomerCity
1	何欢	成都
2	晏婉	北京
3	鞠丁	上海

其中，属性 CustomerID 是客户的唯一标识，用于指定唯一的一个客户。因为客户姓名可能重复，所在城市也有可能重复，所以只有通过一个系统内部编号来唯一标识该实例。

8.1.3　关系集

关系（Relationship）是指多个实体间的相互关联。例如可以定义学生张三和班级 07 级 2 班相关联的关系。这一联系指明张三是 07 级 2 班的学生。关系集（Relationship Set）是同类联系的集合。关系集是 n（$n \geqslant 2$）个实体集上的数学关系，这些实体集不必互异。

实体集的关联称为参与。E-R 模型中关系实体使用菱形来表示，表示在两个实体间的一个关联，如图 8.3 所示为学生实体和课程实体之间的关系。

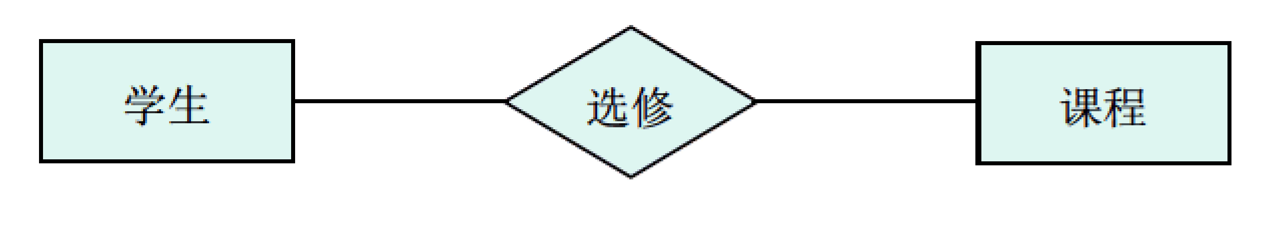


图 8.3　关系表示

例如一个学生张三和另外一个班级实体 07 级 2 班参与到一个关系实例中。实体在关系中的作用称为实体的角色。由于参与一个关系的实体集通常是互异的，所以角色是隐含的并且一般并不指定。但是，当关系的含义需要解释时角色是十分有用的。

关系也可以具有描述性属性。例如学生实体和课程实体之间存在关系，这里将它们之间的关系集称为选课。学生实体通过选课和课程实体建立了关系，选课除了标识出选课的学生和所选的课程外还可以具有选课时间、选课状态等属性。

8.1.4　属性

每个属性都有一个可取值的集合称为该属性的域。CustomerName 属性的域是一定长度的字符串集合。同样 CustomerID 的域是正整数集合。

属性使用椭圆来表示。属性属于哪个实体，则哪个实体使用直线相连，如图 8.4 所示。

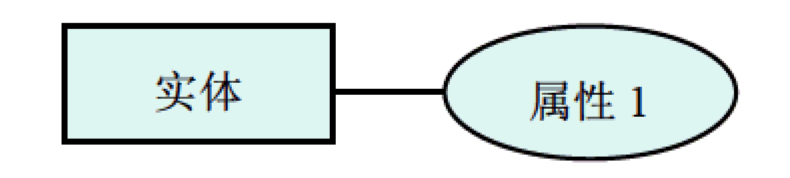


图 8.4　实体与属性表示

从概念上讲，实体集的属性是将实体集映射到域中的函数。在 E-R 模型中属性可以按照如下的属性类型进行划分。

- 简单属性和复合属性。前面例子中说到的大部分属性都是简单属性，也就是说，该属性不能划分成更小的部分。而复合属性可以再划分成更小的部分。例如，CustomerName 属性可以被设计为一个姓属性和一个名属性的复合属性。如果用户希望在某些时候访问整个属性，另一些时候访问属性的一部分，那么设计中使用复合属性是很好的选择。复合属性可以将相关属性聚集起来，使得模型更清晰。
- 单值属性和多值属性。在前面的例子中，每个属性都对应一个特定实体都只有一个单独值。例如对于某个学生，其性别属性、生日属性等都只可能是一个单独的值，这样的属性就是单独属性。而在某些情况下对某个特定实体而言，一个属性可能对应于一组值。假设 Customer 实体集有一个 PhoneNumber 属性，对于不同的客户可能存在 0 个、1 个或多个电话号码，这种情况下 PhoneNumber 属性就被称为多值属性。在需要的情况下，可以对某个多值属性的取值数目进行上下界的限制。
- 派生属性。这类属性的值可以通过别的相关属性或实体派生出来。例如知道了学生实体的出生日期属性，则可以根据该属性派生出当前的年龄属性。又如可以根据 Person 实体的身份证号码属性获得用户的性别属性、出生日期属性等。

当实体的某个属性没有值时使用 Null 值。Null 值可以表示不可用，即该实体的这个属性不存在值。Null 值还可以表示属性值未知。

8.2　关　　系

在现实世界中，事物内部以及事物之间是有联系的，这些联系在信息世界中反映为实体内部的联系和实体之间的关系。实体内部的关系通常是指组成实体的各属性之间的关系。实体之间的关系通常是指不同实体集之间的关系。对于实体间的二元存在一对一、一对多

和多对多 3 种关系。

8.2.1　一对一的关系

如果对于实体集 A 中的每一个实体，实体集 B 中至多只有一个（也可以没有）实体与之联系，反之亦然。则称实体集 A 与实体集 B 具有一对一联系，记为 1：1。根据一侧的关系是否为空，还演变出 0 或一对一的情况。如果用图表示一对一的关系，如图 8.5 所示。

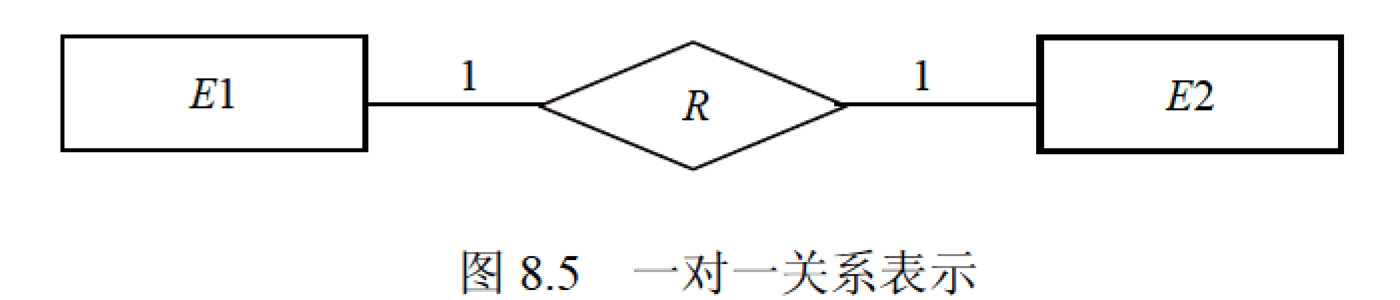


图 8.5　一对一关系表示

例如在学生管理系统中，存在班级实体和教师实体，对于每个班级来说只有一个教师作为班主任，而对于每个教师来说最多只担当一个班级的班主任。在这里就是一对一关系情况。

在 SQL Server 中没有实现真正的一对一的继承方法。可以说，A 表需要 B 表中有一条映射记录，但是当在 B 表中添加记录时必须要在 A 表中有一条匹配记录，这样就产生了矛盾——到底哪张表的记录先产生？

SQL Server 能够处理 0 或一对一的关系实例。这种情况本质上与一对一的情况相同，差别就在于关系的一侧可以有也可以没有匹配记录。前面提到的教师与班级的关系就是 0 或一对一的情况，因为并不是所有教师都担当了班主任，所以一个教师可以对应 0 个班级。

可以在 SQL Server 中通过以下方式强制执行 0 或一对一的关系：

- 一个唯一键或主键与外键约束之间的组合。外键约束可以强制在一张表中至少有一条记录，但不能确定仅有一条记录存在，使用唯一键或主键可以确定只有一条记录。
- 触发器。在两个实体对应的表中都添加触发器。

利用 0 或一对一关系，可以在 SQL Server 中首先在必须的表中插入记录，然后再根据需要在可选表中插入记录。例如可以先向教师表插入数据，然后再向班级表插入数据，插入班级表数据时可以通过外键引用教师表中的数据。

8.2.2　一对多的关系

如果对于实体集 A 中的每一个实体，实体集 B 中有 n 个实体（$n \geq 0$）与之联系，反之，对于实体集 B 中的每一个实体，实体集 A 中至多只有一个实体与之联系，则称实体集 A 与实体集 B 有一对多联系，记为 1：n。如果用图表示一对多的关系，如图 8.6 所示。

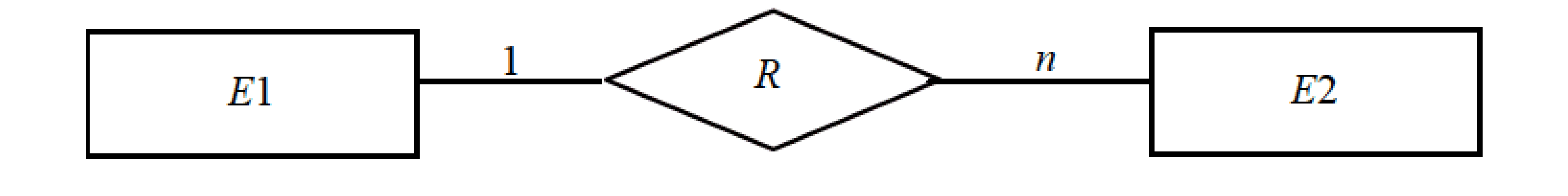


图 8.6　一对多关系表示

一对多关系是平时接触的一种关系，是外键关系的一种形式。例如在学生管理系统中，

一个学生只属于一个班级，但是一个班级中却有多个学生。

另外，还有一种常见的一对多关系是主表和子表的关系。例如一个订单由订单主项和订单明细组成，一个订单只有一份主项，但是却可以有多个明细项，所以在表示订单中是 1 份主项对应多个明细项的一对多关系。各种对账单、发票和报销单都是如此。

在 SQL Server 中，通过两种方法加强这种关系：

- 外键约束。在作为“多”侧关系的表中声明一个外部键约束，而且引用表和列将作为关系的“一”侧。因为必须在外部键的引用列上有主键或唯一性约束，所以假定在引用表中只有一个这样的“一”侧。
- 触发器。实际上，对于早期的 SQL Server 版本，这是引用完整性的唯一选项。实际需要添加两个触发器——一侧关系一个。在关系的“多”侧表中添加一个触发器，并检测该表中被插入或改变的任何行在其所依赖的表中有一个匹配行。然后在另一个表中建立添加、删除和更新触发器——该触发器检查从引用表中删除或更改的记录，确保该数据不成为孤儿。

当然，从目前来说，基本上所有的一对多关系都是靠外键来完成的，使用外键的速度更快而且比触发器方式更容易维护。

以班级和学生的一对多关系建立表，学生表中需要“班级编号”字段，该字段是班级表的主键。在学生表中对班级编号字段建立外键约束，从而确定了每一个学生对应的班级都实际存在，避免了通过学生的班级编号找不到任何班级的情况发生。

一个实体集可以和自己呈现一对多关系，例如对于部门实体，一个部门可以对应多个下级部门，而一个下级部门只能对应一个上级部门，所以部门与部门之间是一对多的关系。

8.2.3　多对多的关系

如果对于实体集 A 中的每一个实体，实体集 B 中有 n 个实体（$n \geqslant 0$）与之联系，反之，对于实体集 B 中的每一个实体，实体集 A 中也有 m 个实体（$m \geqslant 0$）与之联系，则称实体集 A 与实体集 B 具有多对多联系，记为 m：n。若用图表示多对多关系，如图 8.7 所示。

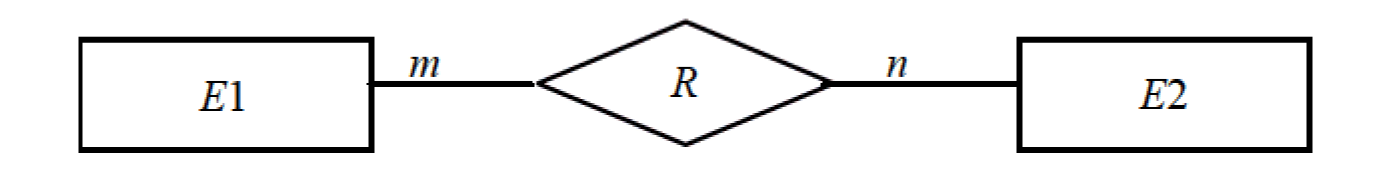


图 8.7　多对多关系表示

例如在学生管理系统中，学生和课程之间的关系是一个学生会选修多门课程，一门课程会有多个学生选修。

SQL Server 中使用表和外键约束不能表示多对多关系，所以必须要用中间表来组织这种关系。在 SQL Server 中并不是所有表都对应这一个实体，有些表的存在只是为了建立这种关系。这种形式的中间表通常被称为链接表或相关表。

同样以学生和课程之间的多对多关系为例，学生实体对应表 Student（主键 StudentID），课程实体对应表 Course（主键 CourseID）。为了建立学生和课程之间的多对多关系，需要建立选课表（ChooseCourse），该表至少含有 CourseID 列和 StudentID 列，可以将这两个列作为联合主键，防止同样的 StudentID 和 CourseID 在该表中出现多次的情况。然后在

ChooseCourse 的 StudentID 列上建立外键约束，指向 Student 表的主键 StudentID，CourseID 列上建立外键约束，指向 Course 表的主键 CourseID。这样通过中间表和外键约束的形式便可在 SQL Server 中实现多对多。

同一对多一样，多对多也存在实体集与自身形成多对多的情况。例如对于人员实体集（Person），人与人直接存在朋友关系，一个人对应多个朋友，而朋友那边也是一个人对应多个朋友。这种情况下需要建立相关表 Friends，其中有列 PersonAID 和 PersonBID 用于表示 A 与 B 直接是朋友关系。

8.3　范　　式

构造数据库必须遵循一定的规则。在关系数据库中，这种规则就是范式。范式是符合某一种级别的关系模式的集合。关系数据库中的关系必须满足一定的要求，即满足不同的范式。目前关系数据库有以下几种范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、Boyce-Codd 范式（BCNF）、第四范式（4NF）、第五范式（5NF）和第六范式（6NF）。满足最低要求的范式是第一范式（1NF）。在第一范式的基础上进一步满足更多要求的称为第二范式（2NF），其余范式依次类推。一般来说，数据库只需满足第三范式（3NF）和 BC 范式就行了。

8.3.1　第一范式

在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。

第一范式的定义为：数据库表中的字段都是单一属性的，不可再分。第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。例如表 8.3 员工信息表，不能将员工信息都放在一列中显示，也不能将其中的两列或多列在一列中显示；员工信息表的每一行只表示一个员工的信息，一个员工的信息在表中只出现一次。简而言之，第一范式就是无重复的列。

表 8.3　符合第一范式的员工信息表

员　工　号	姓　　名	年　　龄	性　　别
00001	小张	23	男
00002	小王	29	女
00003	小庄	30	男

而如表 8.4 所示，表中使用部门编码和员工在部门中的唯一编码来表示员工号，这些编码可以分解为更小的单元，因此是非原子的。这种情况就是不符合第一范式的。

表 8.4 不符合第一范式的员工信息表

员　工　号	姓　　名	年　　龄	性　　别
JW00001	小张	23	男
FA00002	小王	29	女
JW00002	小庄	30	男

除了重复的列以外前面还提到同一列中不能有多个值，例如表 8.5 所示，虽然该表中没有重复的记录，但是对于子女姓名列，一个员工可能对应多个子女，这种一对多的关系不能用一张表来表示；否则是不符合第一范式的。

表 8.5 不符合第一范式的员工信息表

员　工　号	姓　　名	年　　龄	性　　别	子 女 姓 名
00001	小张	23	男	张三
00002	小王	29	女	刘伟，刘毅
00003	小庄	30	男	Null

说明： XML 类型的出现允许在一个列中表示一个集合，这里需要将 XML 列看成一个属性，仍然是符合第一范式的。

8.3.2 第二范式

第二范式（2NF）：数据库表中不存在非关键字段对任一候选关键字段的部分函数依赖（部分函数依赖指的是存在组合关键字中的某些字段决定非关键字段的情况），即所有非关键字段都完全依赖于任意一组候选关键字。

第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或行必须可以被唯一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。如表 8.3 员工信息表中加上了员工号（emp_id）列，因为每个员工的员工编号是唯一的，因此每个员工可以被唯一区分。这个唯一属性列被称为主关键字或主键、主码。

第二范式（2NF）要求实体的属性完全依赖于主关键字。完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。

假定选课关系表为 SelectCourse（学号，姓名，年龄，课程名称，成绩，学分），关键字为组合关键字（学号，课程名称），如表 8.6 所示。

表 8.6 不符合第二范式的SelectCourse表

学　　号	姓　　名	年　　龄	课 程 名 称	成　　绩	学　　分
2303002026	小张	23	大学物理	87	5
2303002027	小曾	23	电路分析	80	6
2303002028	小胡	23	电路分析	98	6

该表中存在如下决定关系：

```
(学号,课程名称)→(姓名,年龄,成绩,学分)
```

这个数据库表不满足第二范式，因为存在如下决定关系：

```
(课程名称)→(学分)
(学号)→(姓名,年龄)
```

即存在组合关键字中的字段决定非关键字的情况。由于不符合第二范式，这个选课关系表会存在如下问题。

① 数据冗余：同一门课程由 *n* 个学生选修，“学分”就重复 *n*–1 次；同一个学生选修了 *m* 门课程，姓名和年龄就重复了 *m*–1 次。

② 更新异常：若调整了某门课程的学分，数据表中所有行的“学分”值都要更新；否则会出现同一门课程学分不同的情况。

③ 插入异常：假设要开设一门新的课程，暂时还没有人选修。这样，由于还没有“学号”关键字，课程名称和学分也无法记入数据库。

④ 删除异常：假设一批学生已经完成课程的选修，这些选修记录就应该从数据库表中删除。但是，与此同时，课程名称和学分信息也被删除了。很显然，这样会导致插入异常。

把选课关系表 SelectCourse 改为如下 3 个表。

- 学生：Student（学号，姓名，年龄）；
- 课程：Course（课程名称，学分）；
- 选课关系：SelectCourse（学号，课程名称，成绩）。

修改为符合第二范式的选课关系表如表 8.7 所示。

表 8.7　符合第二范式的表结构

学号	姓名	年龄
2303002026	小张	23
2303002027	小曾	23
2303002028	小胡	23

学生表

学号	课程名称	成绩
2303002026	大学物理	87
2303002027	电路分析	80
2303002028	电路分析	98

选课关系

课程名称	学分
大学物理	5
电路分析	6

课程表

这样的数据库表是符合第二范式的，消除了数据冗余、更新异常、插入异常和删除异常。另外，所有单关键字的数据库表都符合第二范式，因为不可能存在组合关键字。

8.3.3　第三范式

第三范式（3NF）：在第二范式的基础上，数据表中如果不存在非关键字段对任一候选关键字段的传递函数依赖则符合第三范式。传递函数依赖指的是如果存在“A→B→C”的决定关系，则 C 传递函数依赖于 A。因此，满足第三范式的数据库表应该不存在如下依赖关系：关键字段→非关键字段 x→非关键字段 y。

满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要

求一个数据库表中不包含其他表中已包含的非主关键字信息。例如，存在一个部门信息表，其中每个部门有部门编号、部门名称、部门简介等信息。

那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它；否则就会有大量的数据冗余。简而言之，第三范式就是属性不依赖于其他非主属性。

假定学生关系表为 Student（学号，姓名，年龄，所在学院，学院地点，学院电话），关键字为单一关键字“学号”，如表 8.8 所示。

表 8.8　不符合第三范式的学生关系表

学　　号	姓　　名	年　　龄	所 在 学 院	学 院 地 点	学 院 电 话
230300201	张三	23	微固学院	3 系大楼	833323
230300302	李四	24	微固学院	3 系大楼	833323
230500201	王五	23	光电学院	5 系大楼	833348

其中存在如下决定关系：

```
(学号)→(姓名,年龄,所在学院,学院地点,学院电话)
```

这个数据库是符合第二范式的，但是不符合第三范式，因为存在如下决定关系：

```
(学号)→(所在学院)→(学院地点,学院电话)
```

即存在非关键字段“学院地点”、“学院电话”对关键字段“学号”的传递函数依赖。它也会存在数据冗余、更新异常、插入异常和删除异常的情况。

① 数据冗余：同一个学院对应 n 个学生，学院地点和学院电话就重复了 n–1 次。

② 更新异常：若调整了某学院的电话，数据表中所有相关行的“学院电话”字段的值都要更新；否则会出现同一个学院对应多个电话的情况。

③ 插入异常：假设要开设一个新的学院，而现在学院中还没有招收学生。这样，由于还没有“学号”关键字，学院地点和学院电话也无法记录入数据库。

④ 删除异常：假设一个学院已经被取消，应该从数据库表中删除。但是，与此同时，学生信息也被删除了。

要避免这些异常可以把学生关系表分为如下两个表：

- ❑ 学生：（学号，姓名，年龄，所在学院）；
- ❑ 学院：（学院，地点，电话）。

如表 8.9 所示。

表 8.9　符合第三范式的学生关系表

学号	姓名	年龄	所在学院
230300201	张三	23	微固学院
230300302	李四	24	微固学院
230500201	王五	23	光电学院

学生表

学院	学院地点	学院电话
微固学院	3 系大楼	833323
光电学院	5 系大楼	833348

学院表

这样的数据库表是符合第三范式的，消除了数据冗余、更新异常、插入异常和删除

异常。

8.3.4 Boyce-Codd 范式

Boyce-Codd 范式（BCNF）：在第三范式的基础上，数据库表中如果不存在任何字段对任一候选关键字段的传递函数依赖，则符合第三范式。在有些书中，Boyce-Codd 范式被当做第三范式的一种变化形式，而没有单独提出。

假设仓库管理关系表为 StorehouseManage（仓库 ID，存储物品 ID，管理员 ID，数量），且一个管理员只在一个仓库工作；一个仓库可以存储多种物品，则这个数据库表中存在如下决定关系：

```
(仓库 ID,存储物品 ID)→(管理员 ID,数量)
(管理员 ID,存储物品 ID)→(仓库 ID,数量)
```

所以，（仓库 ID，存储物品 ID）和（管理员 ID，存储物品 ID）都是 StorehouseManage 的候选关键字，表中的唯一非关键字段为数量，它是符合第三范式的。但是，由于存在如下决定关系：

```
(仓库 ID)→(管理员 ID)
(管理员 ID)→(仓库 ID)
```

即存在关键字段决定关键字段的情况，所以其不符合 BCNF 范式。它会出现如下异常情况。

- 删除异常：当仓库被清空后，所有“存储物品 ID”和“数量”信息被删除的同时，“仓库 ID”和“管理员 ID”信息也被删除了。
- 插入异常：当仓库没有存储任何物品时，无法给仓库分配管理员。
- 更新异常：如果仓库换了管理员，则表中所有行的管理员 ID 都要修改。

把仓库管理关系表分解为以下两个关系表。

- 仓库管理：StorehouseManage（仓库 ID，管理员 ID）。
- 仓库：Storehouse（仓库 ID，存储物品 ID，数量）。

这样的数据库表是符合 BCNF 范式的，消除了删除异常、插入异常和更新异常。

8.3.5 其他范式

在 SQL Server 数据库设计中，能够满足 BCNF 范式要求已经是很好的情况了。更高的第四范式、第五范式乃至于第六范式都主要用于学术研究，很少在实际项目中使用，这里就做一个简单介绍。

第四范式（4NF）：设 R 是一个关系模式，D 是 R 上的多值依赖集合。如果 D 中成立非平凡多值依赖 X→Y 时，X 必是 R 的超键，那么称 R 是第四范式（4NF）。满足第四范式则必然满足 BC 范式。第四范式设法处理多值依赖性问题。这是独立行没有列依赖于除主键和整个主键的情况。但是，这是非常少有的主键中的列单独依赖主键中其他列的情形，而且通常不能解决实际问题。因此，在数据库设计中通常被忽略了。

第五范式（5NF）：如果关系模式 R 中的每一个连接依赖，都是由 R 的候选键所蕴涵，

称 R 是第五范式。第五范式又叫做投影—连接范式，用于处理非丢失和丢失分解。本质上，这是可以分解某种关系的情形，但此时不能再从逻辑上把它重新组织成原始形式。这种情况非常少，在很大程度上都是学术研究的问题。

第六范式（6NF）又叫做域键范式，只有消除了修改异常可能性才能产生这种标准形式。这是一种非常理想化的范式，很少被使用。

范式可以避免数据冗余，减少数据库的空间，减轻维护数据完整性的麻烦，但是操作难。因为需要联系多个表才能得到所需要的数据，而且越高的范式性能就会越差，所以要权衡是否使用更高范式是比较麻烦的。

8.4　数据库建模

到目前已经对实体-关系模型和范式做了详细的介绍，本节将主要讲解如何使用这些基础知识进行数据库建模。

8.4.1　E-R 图

数据库设计同软件开发一样，首先要做的就是需求分析。假设现在需要做一个学生管理系统，具体需求分析如下所述。

1. 功能需求

- 学生信息管理：实现对学生信息的添加、修改、删除功能。
- 班级信息管理：实现对班级信息的添加、修改、删除功能。
- 课程信息管理：实现对课程信息的添加、修改、删除功能。
- 专业信息管理：实现对专业信息的添加、修改、删除功能。
- 系信息管理：实现对系信息的添加、修改、删除功能。
- 成绩信息管理：实现对学生成绩的添加、修改、删除功能。
- 数据查询：包括学生基本信息的查询和学生成绩查询等功能。

2. 数据需求

- 学生：包括学号，姓名，性别，出生日期，联系方式。
- 班级：包括班级编号，班级名称。
- 课程：包括课程编号，课程名称，周学时，课程性质（考试/考查），类型（公共基础/专业基础/专业课/公共选修/专业选修）。
- 专业：包括专业编号，专业名称。
- 系：包括系编号，系名称，系办地址。

3. 其他

- 同一个专业所开设的课程相同。
- 学生学习某门课程都会有成绩记录。

在了解了需求后接下来就是需要画 E-R 图了。前面已经讲到在 E-R 模型中，矩形表示实体、椭圆表示属性、菱形表示关系。就像软件开发中类图的作用和地位一样，E-R 图将是数据库设计中最初也是最重要的表示工具。

如图 8.8 所示为学生管理系统的 E-R 图。

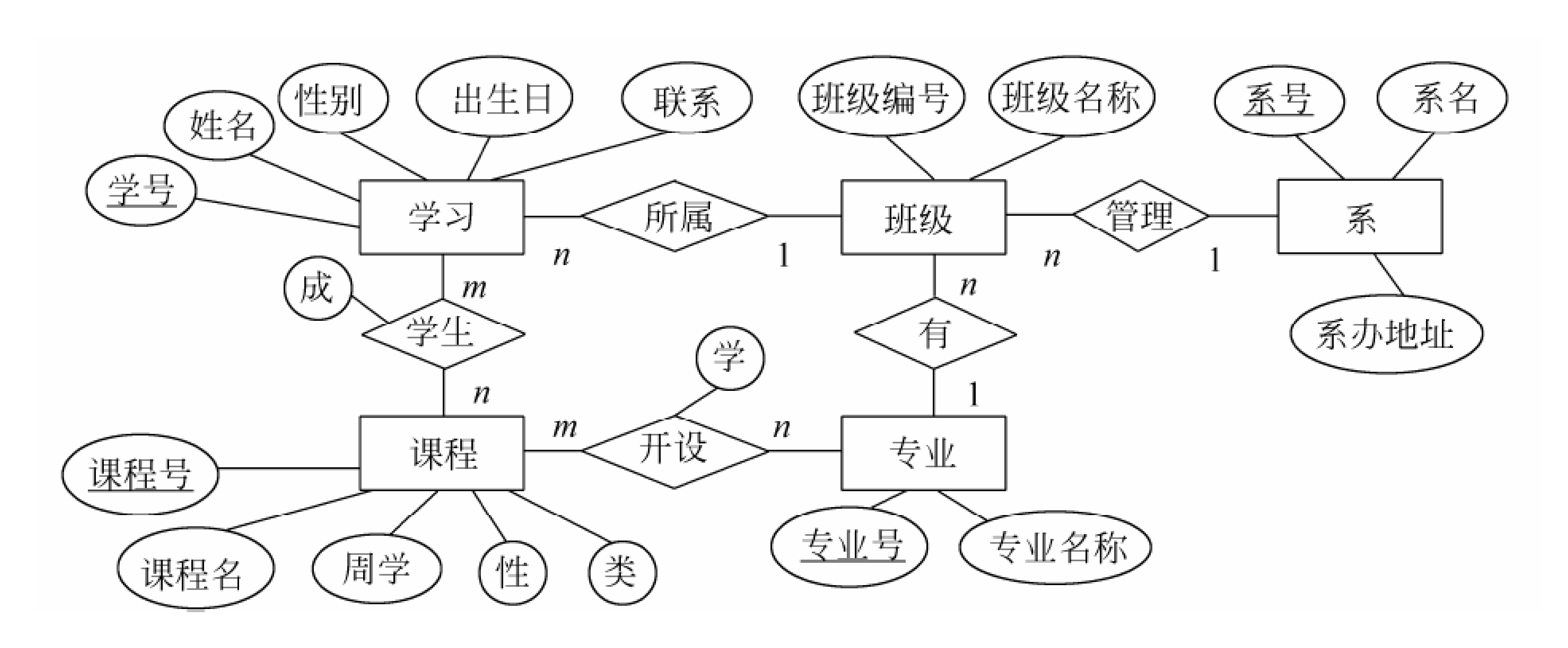


图 8.8　选课系统 E-R 图

在这个系统中主要包含了 5 个实体：系、专业、班级、学生和课程。一个系对应多个班级，一个班级对应多个学生，一个学生可以选多门课程同时一门可以被多个学生选修。一个专业对应多个班级，一个专业开设了多门课程同时一门课程也可以在多个专业中开设，这就是分析出的实体之间的关系。在明确了具体的实体和实体之间的关系后只需要将实体的属性补充完整即可完成整个 E-R 图。这样看起来似乎很简单，但是真正复杂的是整个分析过程，尤其是实体关系的确定和实体属性的确定，同时还要考虑到范式问题。

8.4.2　关系图

在好的数据库设计中，实体关系图是一个非常重要的工具。在 E-R 图中表示为多对多的关系将在关系图中转换为使用中间表和一对多的关系。如图 8.9 所示为某数据库的关系图。

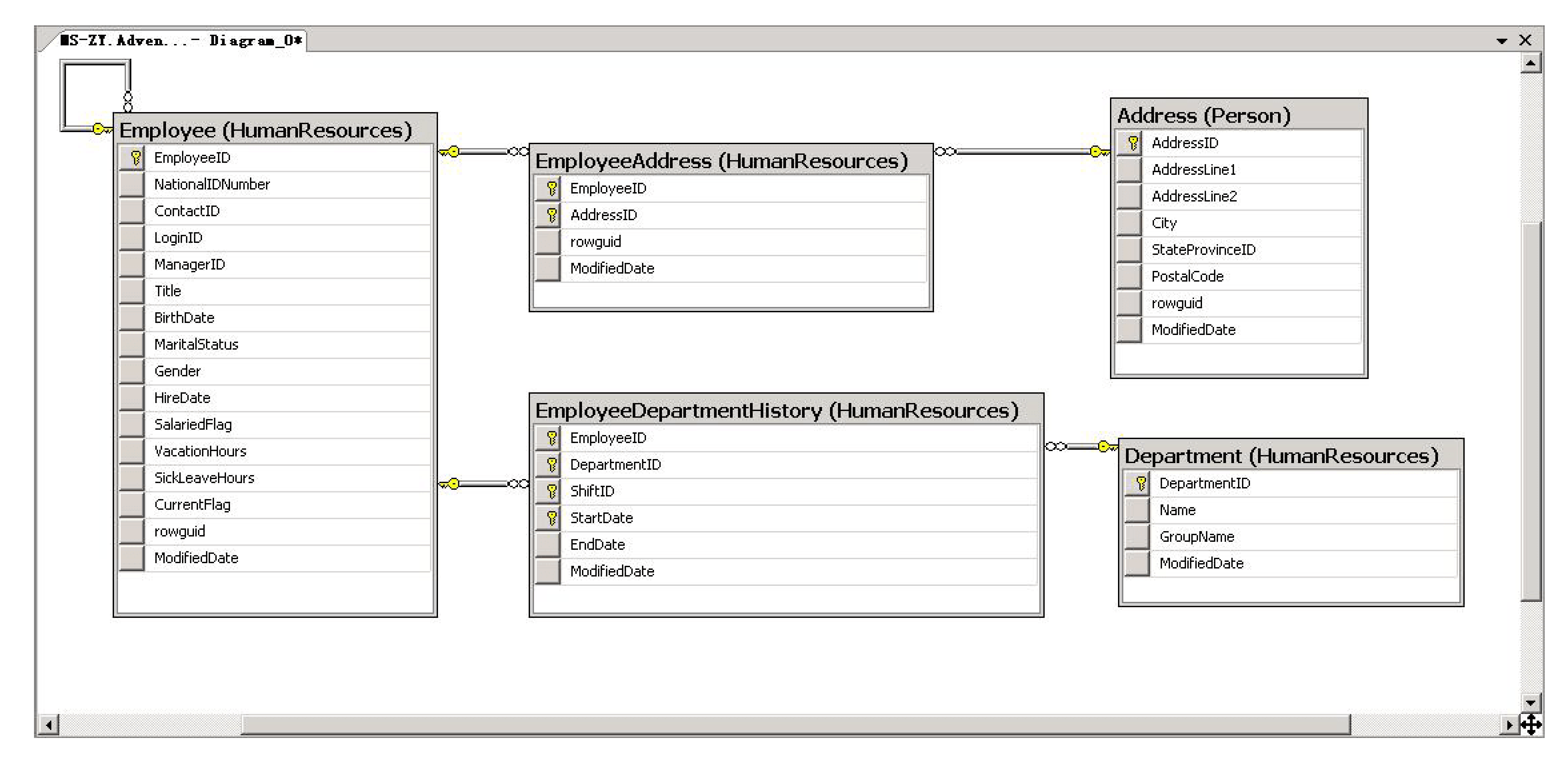


图 8.9　关系图

在 SQL Server 中可以创建数据库关系图。数据库关系图以图形方式显示数据库的结构。使用数据库关系图可以创建和修改表、列、关系和键。此外，还可以修改索引和约束。在 SQL Server 中，使用数据库关系图的主要操作如下所述。

（1）在 SSMS 的对象资源管理器中展开需要画关系图的数据库节点，在“数据库关系图”节点上右击，选择“新建数据库关系图”选项即可打开数据库关系图设计窗口。同时系统还将弹出“添加表”对话框，该对话框中列出了当前数据库下所有的表，如图 8.10 所示。

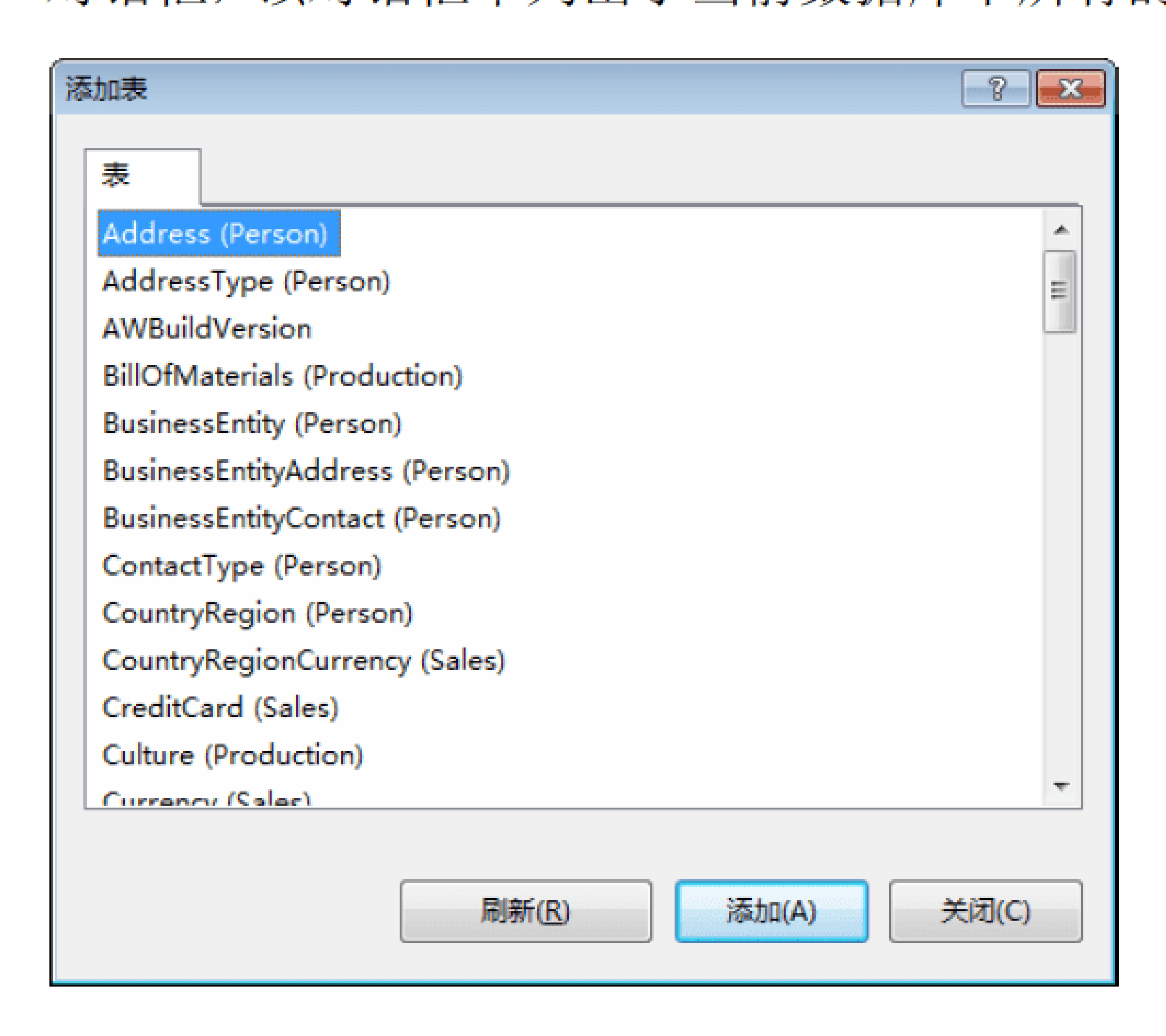


图 8.10　“添加表”对话框

（2）该对话框允许选择在关系图中包含哪些表。选中需要添加的表（可以多选），然后单击“添加”按钮，系统就会将选中的表添加到关系图中。不需要再添加时，可以单击“关闭”按钮回到关系图展示主界面，如图 8.11 所示。

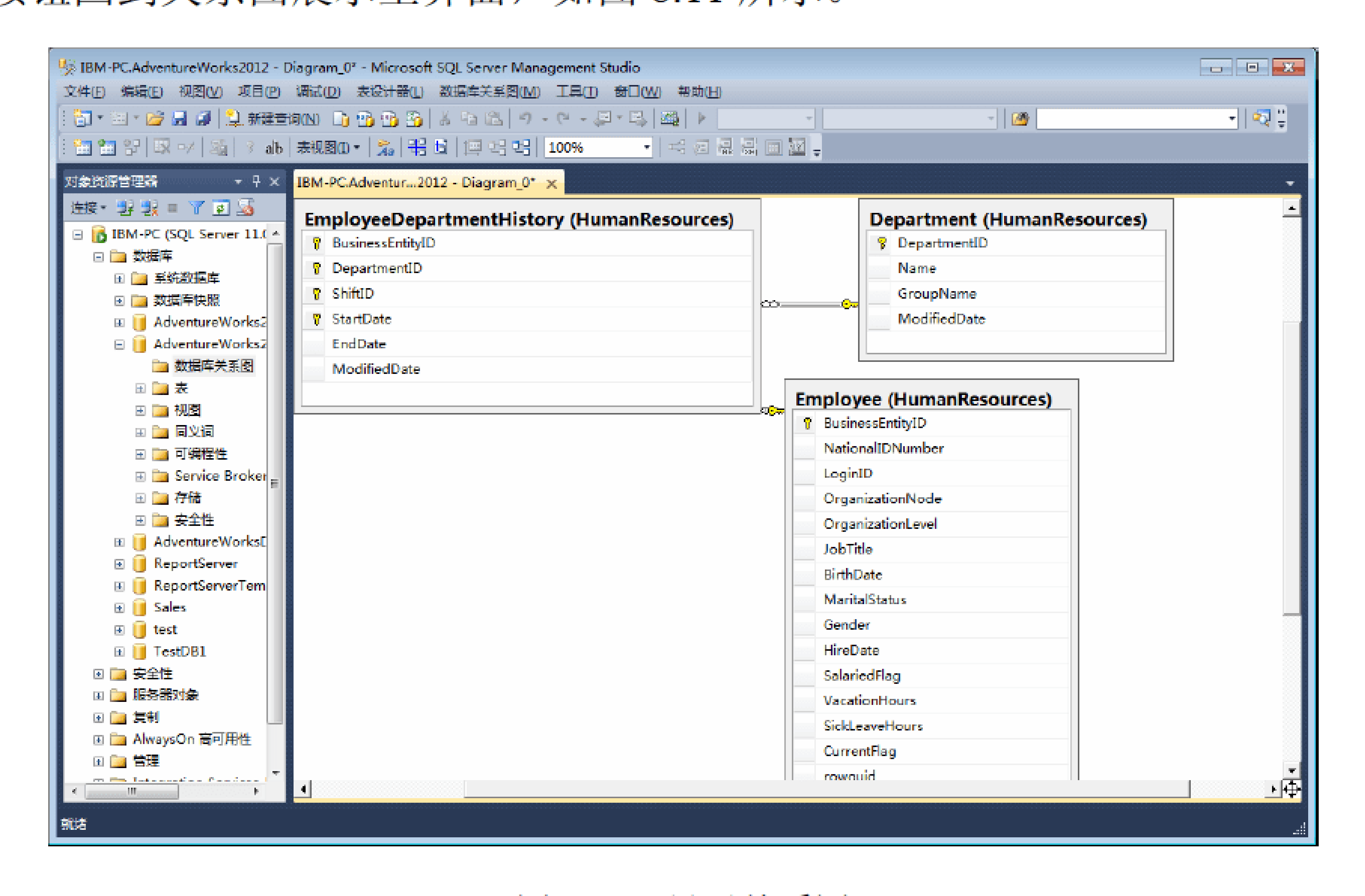


图 8.11　显示关系图

由于关系图中内容较多，所以笔者在此调整了缩放比例以显示全部的对象，在实际使用中用户可以根据自己的需要在 SSMS 中放大或缩小关系图。在关系图显示界面中，可以

通过弹出的快捷菜单或者 SSMS 提供的菜单和工具栏来创建和修改表级相关属性。关于关系图的具体使用方法将在接下来的章节中进行讲解。

8.5　使用 PowerDesigner 进行建模

在数据库建模工具市场中，PowerDesigner（简称 PD）是世界上市场占有率最高的一款建模工具。本节将简要介绍如何使用 PowerDesigner 进行数据库建模。

8.5.1　PowerDesigner 简介

PowerDesigner 作为一款建模产品提供了一个完整的建模解决方案，业务或系统分析人员、设计人员、数据库管理员 DBA 和开发人员可以对其裁剪以满足他们的特定需要。PowerDesigner 使用模块化的结构，这样可方便灵活地购买和扩展，从而使开发单位根据其项目的规模和范围来使用他们所需要的工具。

PowerDesigner 是最具集成特性的设计工具集，用于创建高度优化和功能强大的数据库，数据仓库和数据敏感的组件。

使用 PowerDesigner 不仅加速了开发的过程，也向最终用户提供了管理和访问项目信息的一个有效的结构。PD 非常适用于模型驱动开发，允许设计人员不仅创建和管理数据的结构，而且开发和利用数据的结构针对领先的开发工具环境快速地生成应用对象和数据敏感的组件。开发人员在 PD 中设计了概念模型和物理模型并自动生成数据库后便可使用数据库模型生成程序源代码，基于 ORM 进行快速应用开发。应用对象生成有助于在整个开发生命周期提供更多的控制和更高的生产率。

PowerDesigner 不仅仅是一个数据库建模工具，而且是一个功能强大而使用简单工具集，它提供了一个复杂的交互环境，支持开发生命周期的所有阶段，从处理流程建模到对象和组件的生成。PowerDesigner 产生的模型和应用可以不断地增长，以适应用户的不断变化和日益复杂的需求。

PowerDesigner 包含 6 个紧密集成的模块，允许个人和开发组的成员以合算的方式最好地满足它们的需要。这 6 个模块及说明如下所述。

- PowerDesigner ProcessAnalyst：用于数据发现。
- PowerDesigner DataArchitect：用于双层，交互式的数据库设计和构造。
- PowerDesigner AppModeler：用于物理建模和应用对象及数据敏感组件的生成。
- PowerDesigner MetaWorks：用于高级的团队开发，信息的共享和模型的管理。
- PowerDesigner WarehouseArchitect：用于数据仓库的设计和实现。
- PowerDesigner Viewer：用于以只读的、图形化方式访问整个企业的模型信息。

8.5.2　PowerDesigner 支持的模型

PowerDesigner 支持的 4 种模型文件及其说明如下所述。

- 概念数据模型（CDM）：CDM 表现数据库的全部逻辑的结构，与任何的软件或数据存储结构无关。一个概念模型经常包括在物理数据库中仍然不实现的数据对象。

它给运行计划或业务活动的数据一个正式表现方式。

- ❑ 物理数据模型（PDM）：PDM 叙述数据库的物理实现。在 PDM 中需要考虑真实的物理实现的细节。PDM 需要确定具体的数据库系统，不同的 DBMS 有着不同的物理数据模型。
- ❑ 面向对象模型（OOM）：一个 OOM 包含一系列包、类、接口，以及它们的关系。这些对象一起形成所有的（或部分）一个软件系统的逻辑设计视图的类结构。一个 OOM 本质上是软件系统的一个静态的概念模型。可以使用 PowerDesigner 建立面向对象模型（OOM），然后就能通过 PD 自动产生 Java 源文件、C#源文件或者 PowerBuilder 文件。
- ❑ 业务程序模型（BPM）：BPM 描述业务的各种不同内在任务和内在流程，而且客户如何以这些任务和流程互相影响。BPM 是从业务合伙人的观点来看业务逻辑和规则的概念模型，使用一个图表描述程序、流程、信息和合作协议之间的交互作用。

这其中概念数据模型和物理数据模型是在数据库建模中最常用的，另外两个模型主要用于程序设计上的面向对象分析和业务分析。

8.5.3　建立概念模型

概念模型类似于前面讲到的 E-R 图，不同之处是实体的属性在概念模型中是表示在实体矩形内部的，而不是以椭圆形来表示。目前市面上 PowerDesigner 最新版本为 15.1 版，由于其软件都是收费的，这里选择 15 天的试用版。以学生管理系统为例，在 PowerDesigner 中建立概念模型的操作步骤如下所述。

（1）打开 PowerDesigner，选择 File 菜单下的 New 命令，打开新建模型对话框，如图 8.12 所示。

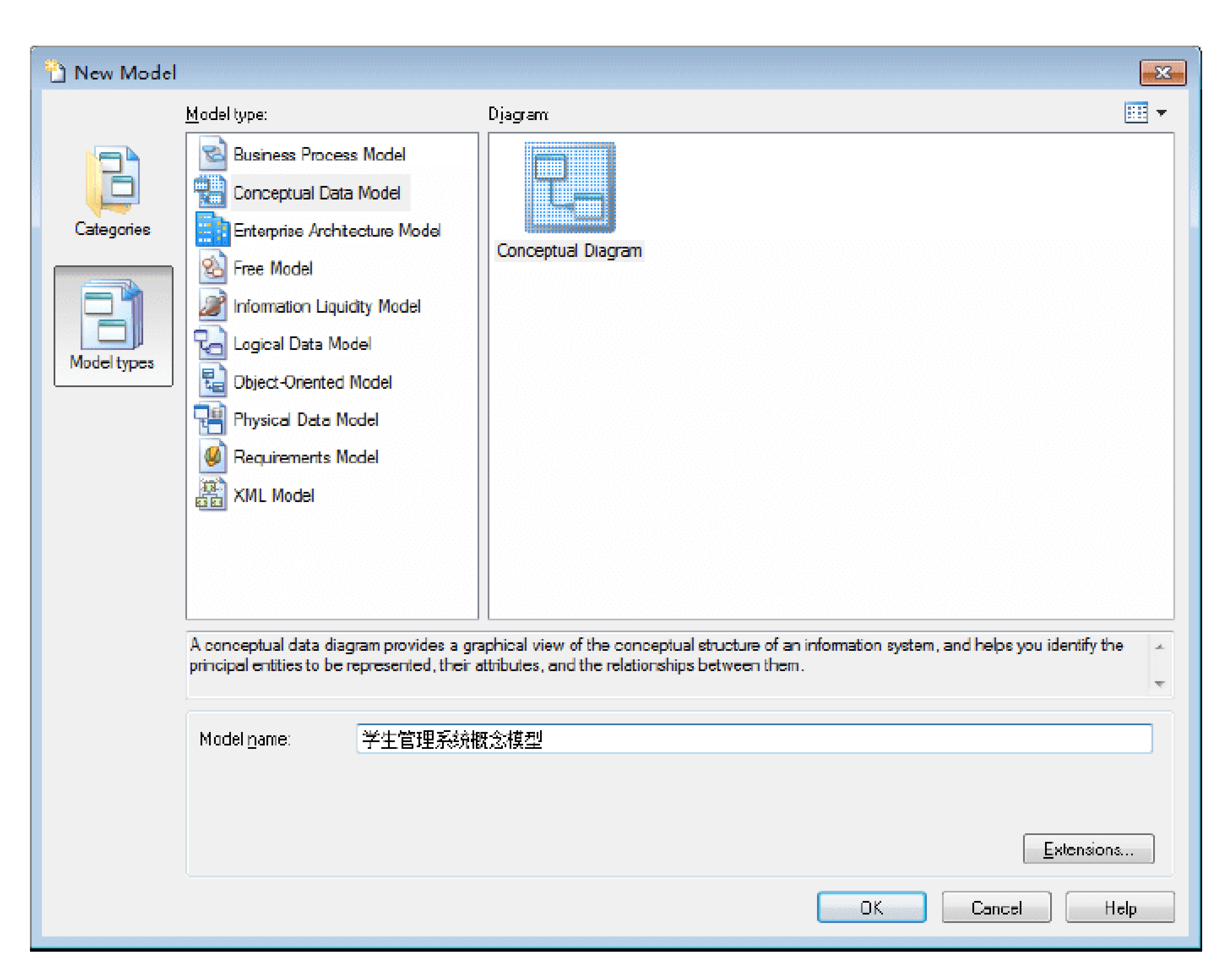


图 8.12　新建模型对话框

（2）在其中选择 Conceptual Data Model，然后在 Model name 文本框中输入模型名“学生管理系统概念模型”，然后单击“确定”按钮，进入模型设计界面，如图 8.13 所示。

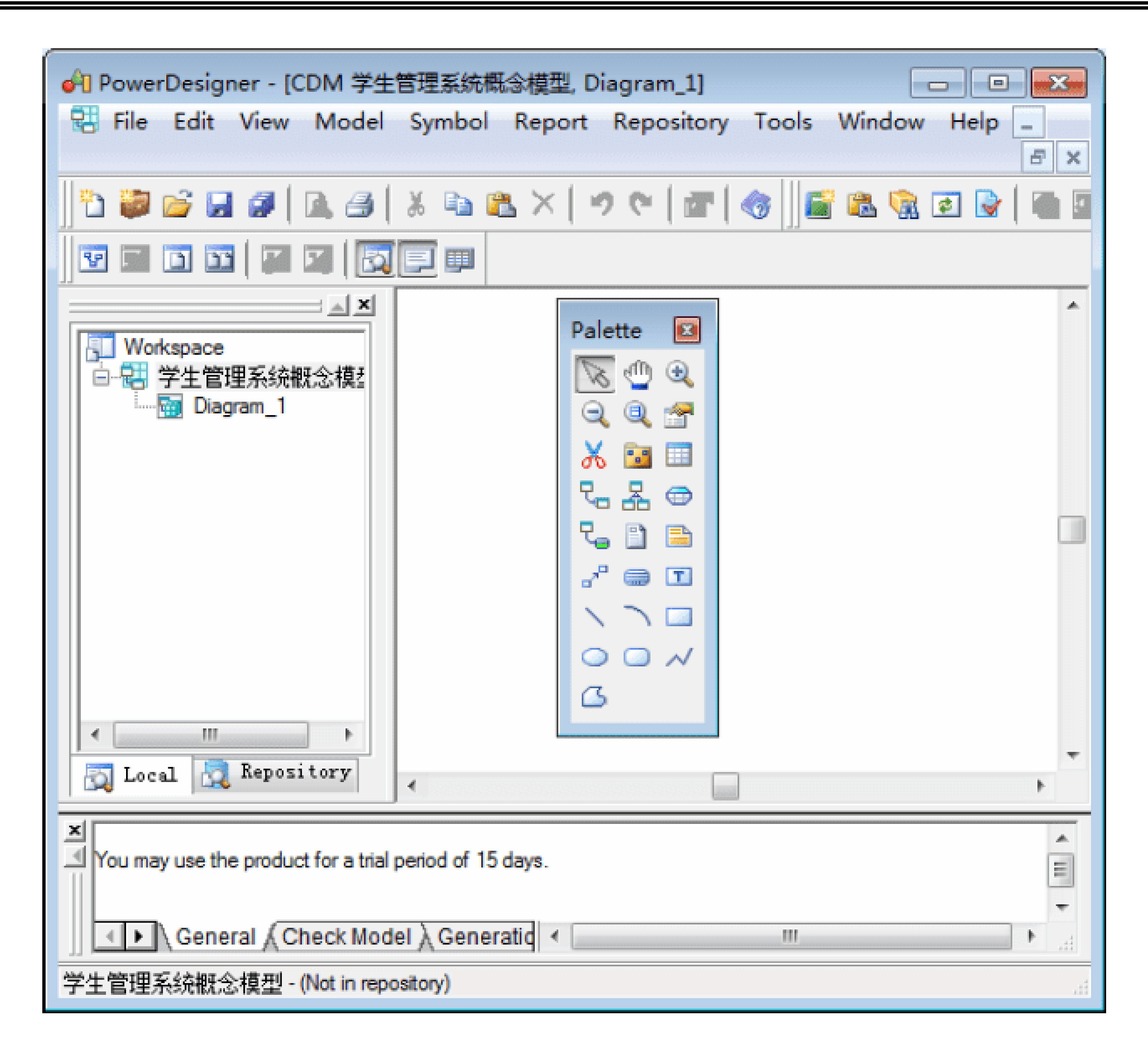
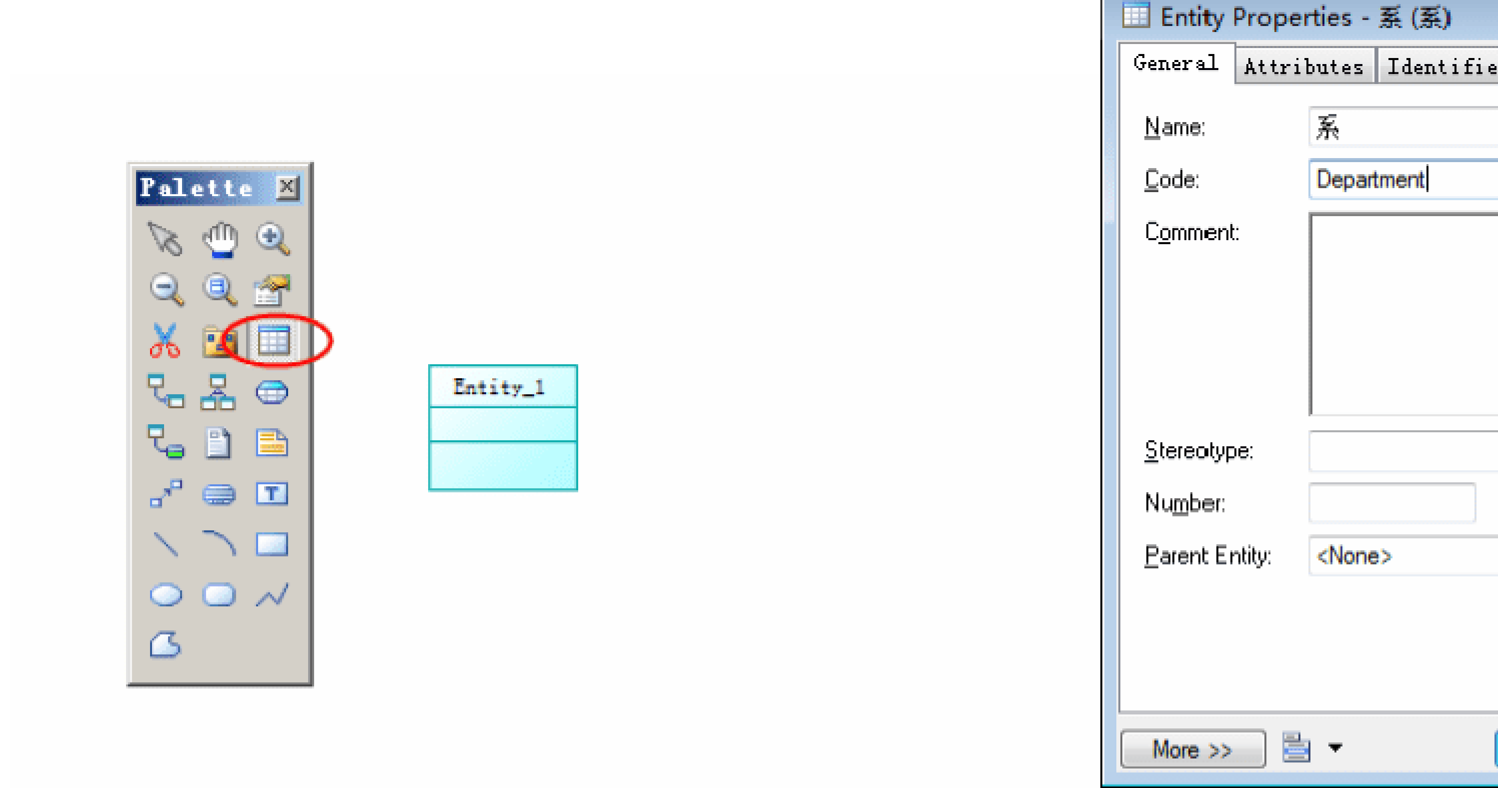


图 8.13　模型设计界面

（3）单击悬浮工具栏中的 Entity 按钮，然后再在主设计面板中单击一次，系统将会在主设计面板中增加一个实体，如图 8.14 所示。

（4）单击悬浮工具栏的指针按钮（Pointer），然后双击主设计面板中的实体矩形，系统将弹出该实体的属性对话框，在其中将实体名由默认的 Entity_1 修改为系，实体的编码修改为系的英文单词 Department，如图 8.15 所示。



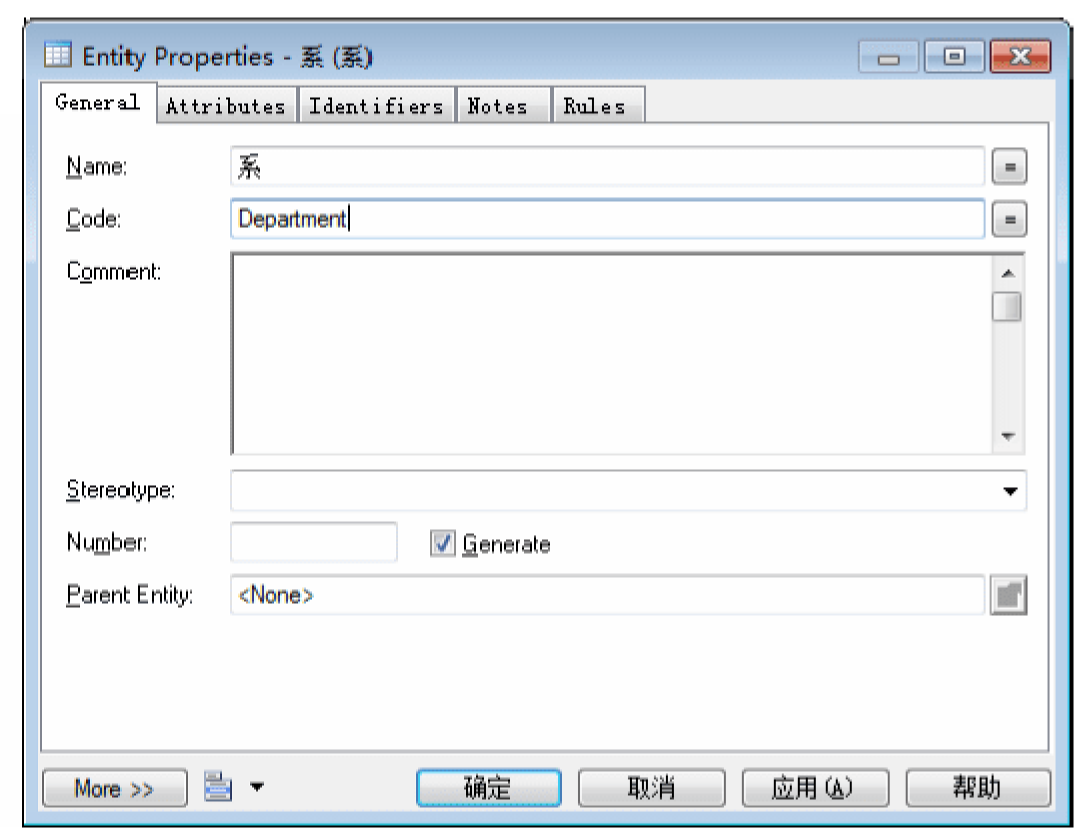


图 8.14　新建实体　　　　图 8.15　修改实体名称和编码



说明： 实体名称是便于建模时阅读和理解的名称，而实体编码则是在最终生成数据库时实体对应的表名。

（5）在属性对话框中选择 Attributes 属性标签，切换到实体属性选项卡，在其中可以设置实体的属性，为系实体添加属性系编号、系名称和系办地址，如图 8.16 所示。

（6）在属性对话框中不但要设置实体的名称和编码，也要设置属性的数据类型。单击系编号行 DataType 列中的按钮，系统将弹出标准数据类型选择对话框，如图 8.17 所示。

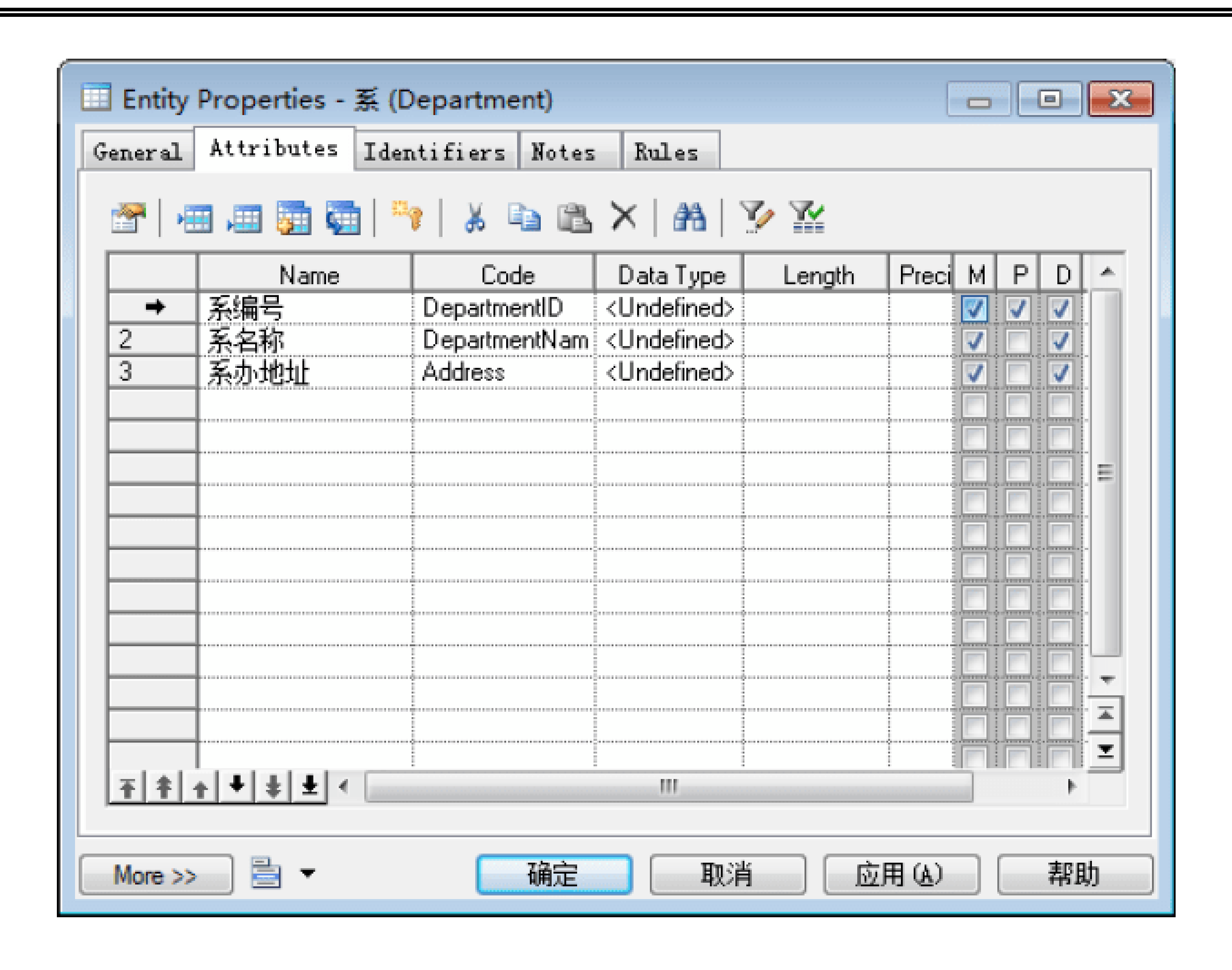


图 8.16　设置实体属性

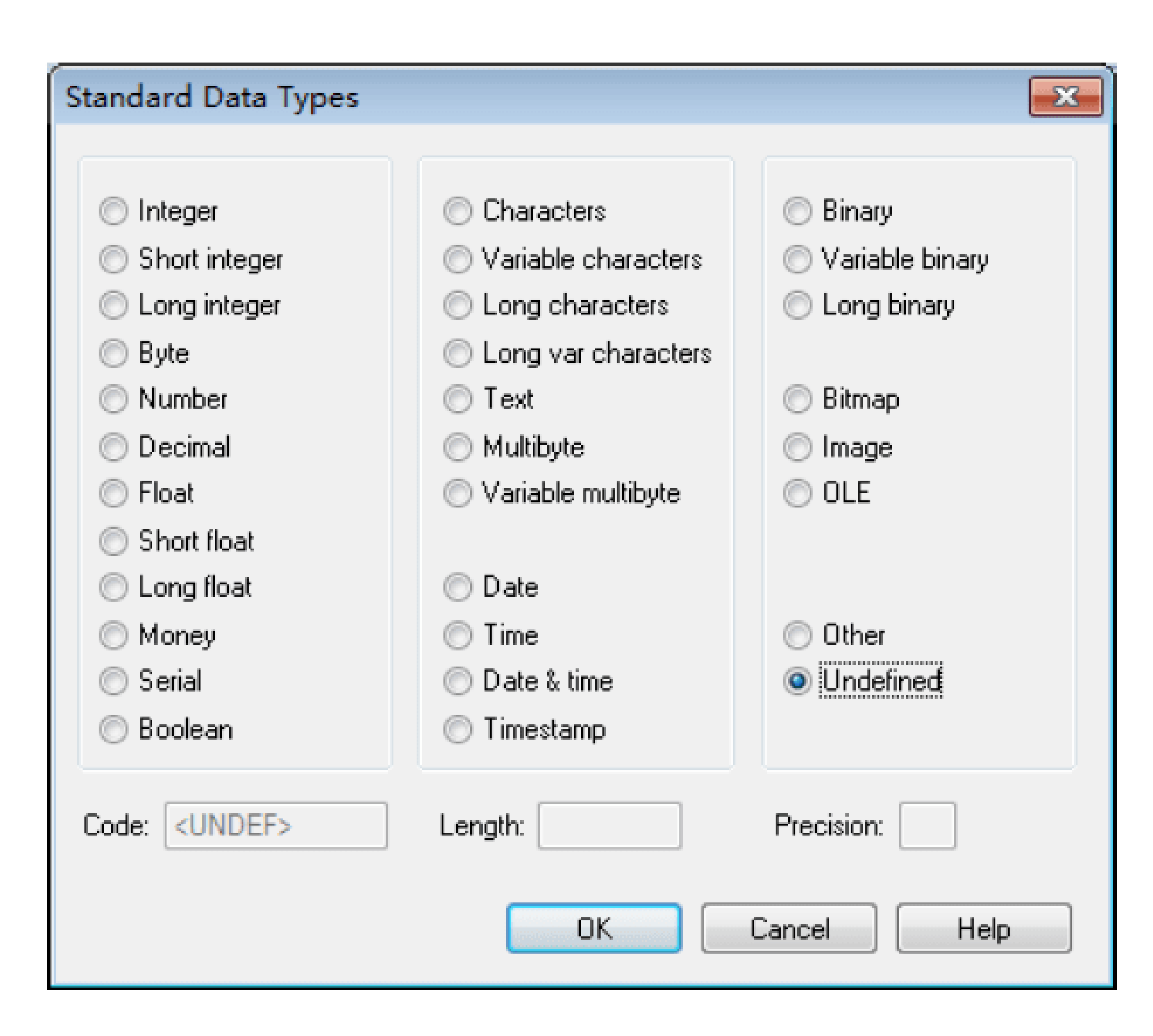


图 8.17　标准数据类型对话框

关于数据类型，笔者在前面的章节中已经做了详细介绍，所以此处就不再讲解。系编号是整数型，所以可以选择 Integer 单选按钮，Length 文本框是用于设置数据类型长度的，一般在字符串类型中使用。

（7）单击 OK 按钮回到实体属性窗口，用同样的方法为系名称设置 Long var characters 类型，长度为 20，系办地址设置为 Long var characters 类型，长度为 50。

（8）除了数据类型外，还有一个重要的设置就是该属性是否必须有值，是否是实体的主键，这些通过后面的 M 列和 P 列复选框来表示。选中 M 列表示不能为空，选中 P 列表示该属性为实体的主键。这些里设置所有属性都不能为空，系编号属性为系实体的主键，选择完成后实体属性对话框如图 8.18 所示。

（9）单击“确定”按钮完成系实体的设置。用同样的方法添加并设置其他实体。添加完所有实体后如图 8.19 所示。

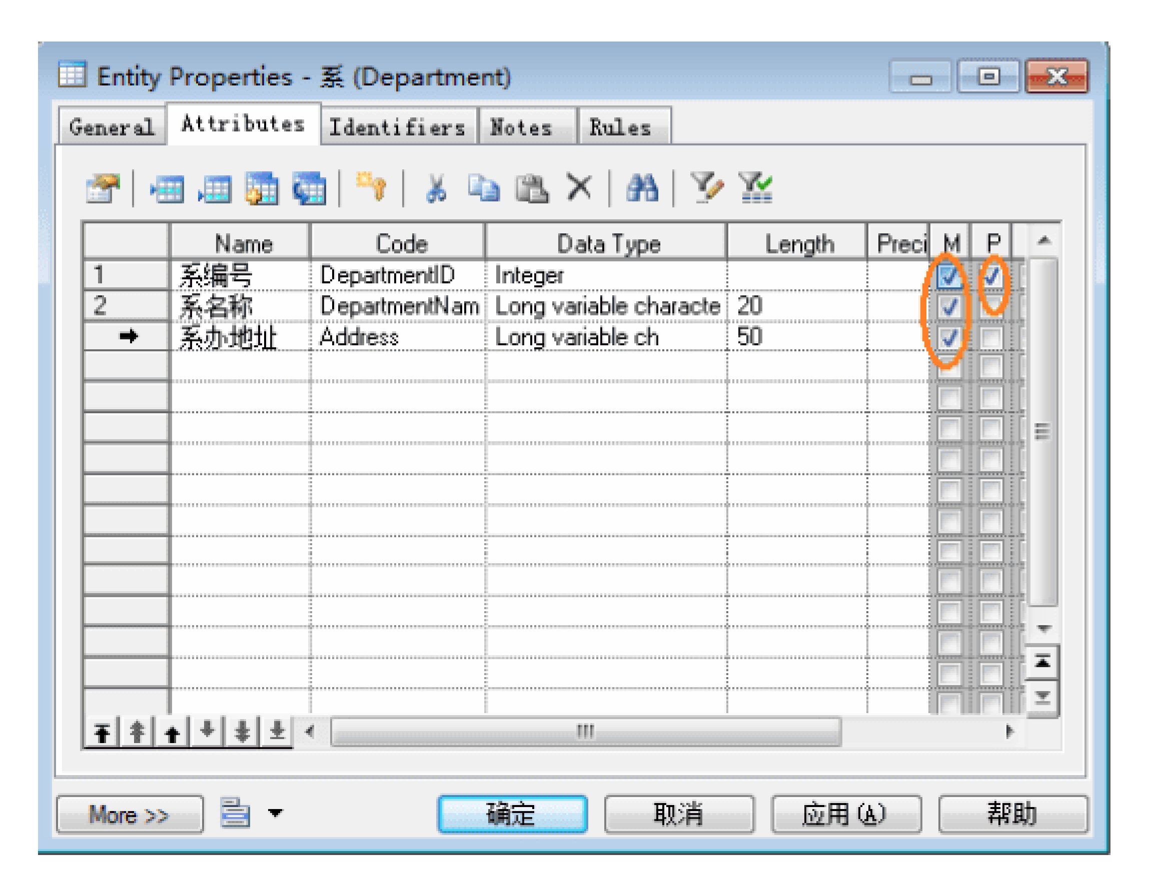


图 8.18　完整的系实体属性设置

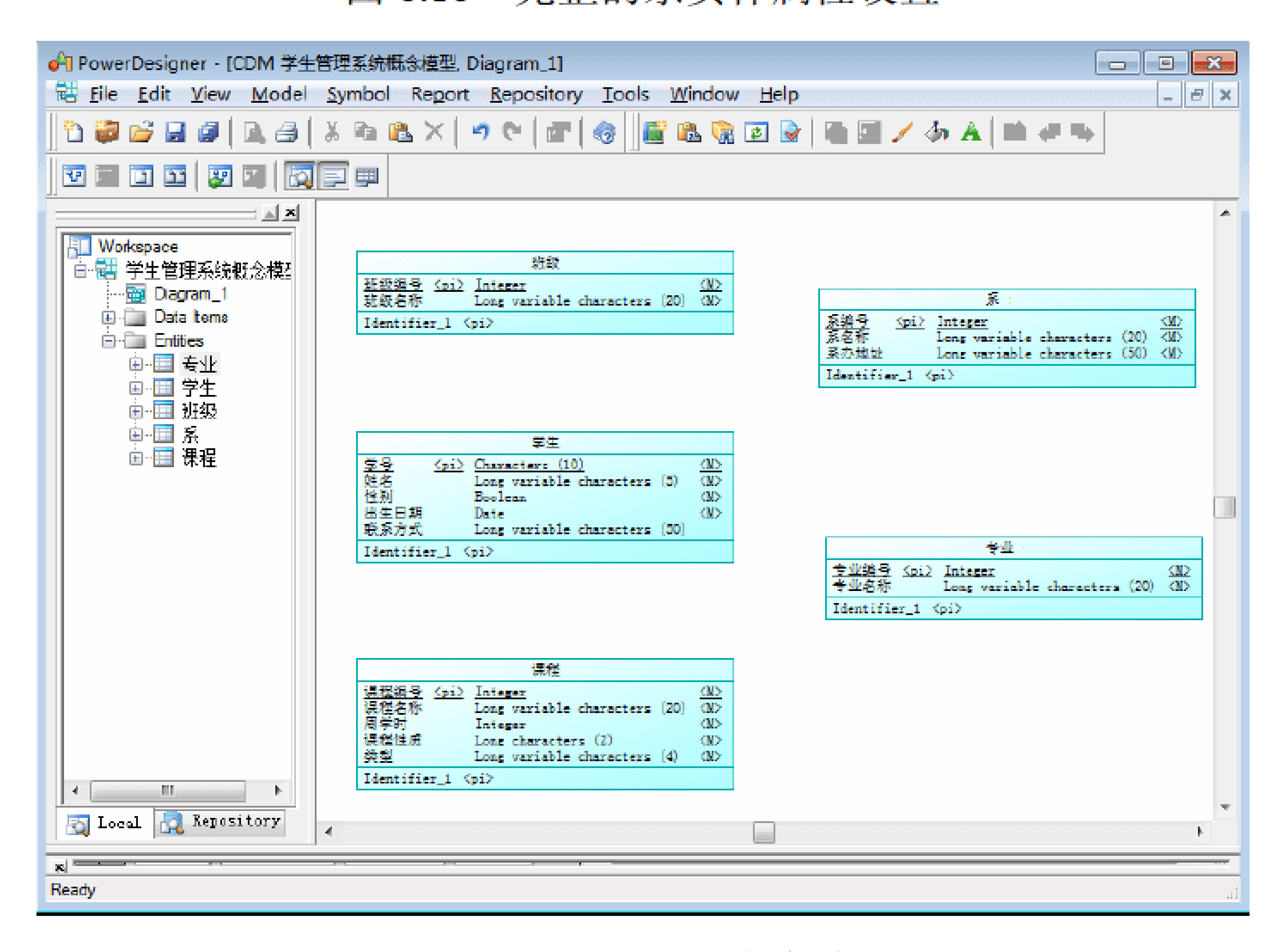


图 8.19　设置所有实体

注意：在概念模型中所有实体的属性名和编码不能重复。如果在 student 实体中设置了 name 属性，那么在 course 实体中就不能再使用 name，而应该使用 coursename 属性，以避免属性名的重复。

（10）在设置完成所有实体后就需要添加实体之间的关系。以系和班级为例，系和班级之间存在着一对多的关系。单击浮动工具栏中的 Relationship 按钮，然后在主设计面板中的系实体上按下鼠标左键，将鼠标拖动到班级实体上，此时系统将会为系实体和班级实体建立关系，如图 8.20 所示。

（11）单击浮动工具栏的指针按钮，然后再双击 Relationship_1 文字，此时系统会弹出该关系的属性对话框，如图 8.21 所示。按照 E-R 图，这里将该关系命名为管理。

（12）选择 Cardinalities 标签，切换到关系类型设置界面，该界面主要用于设置关系是一对一、一对多还是多对多类型。这里系和班级是一对多关系，所以选择 One-Many 单选按钮，如图 8.22 所示。除了设置一对多关系外还可以设置实体是否是必须的，如系实体可以对应 0.n 个班级，而班级对应的系必须存在而且必须是一个。

（13）单击“确定”按钮回到主设计界面。用同样的方法按照 E-R 中给出的关系为其他实体添加关系。添加完成关系后的概念模型如图 8.23 所示。

此时概念模型已经建立完成，在概念模型中，除了“学习”这个多对多关系的属性未

表示外，其他所有 E-R 图中的信息都已经表示出来。

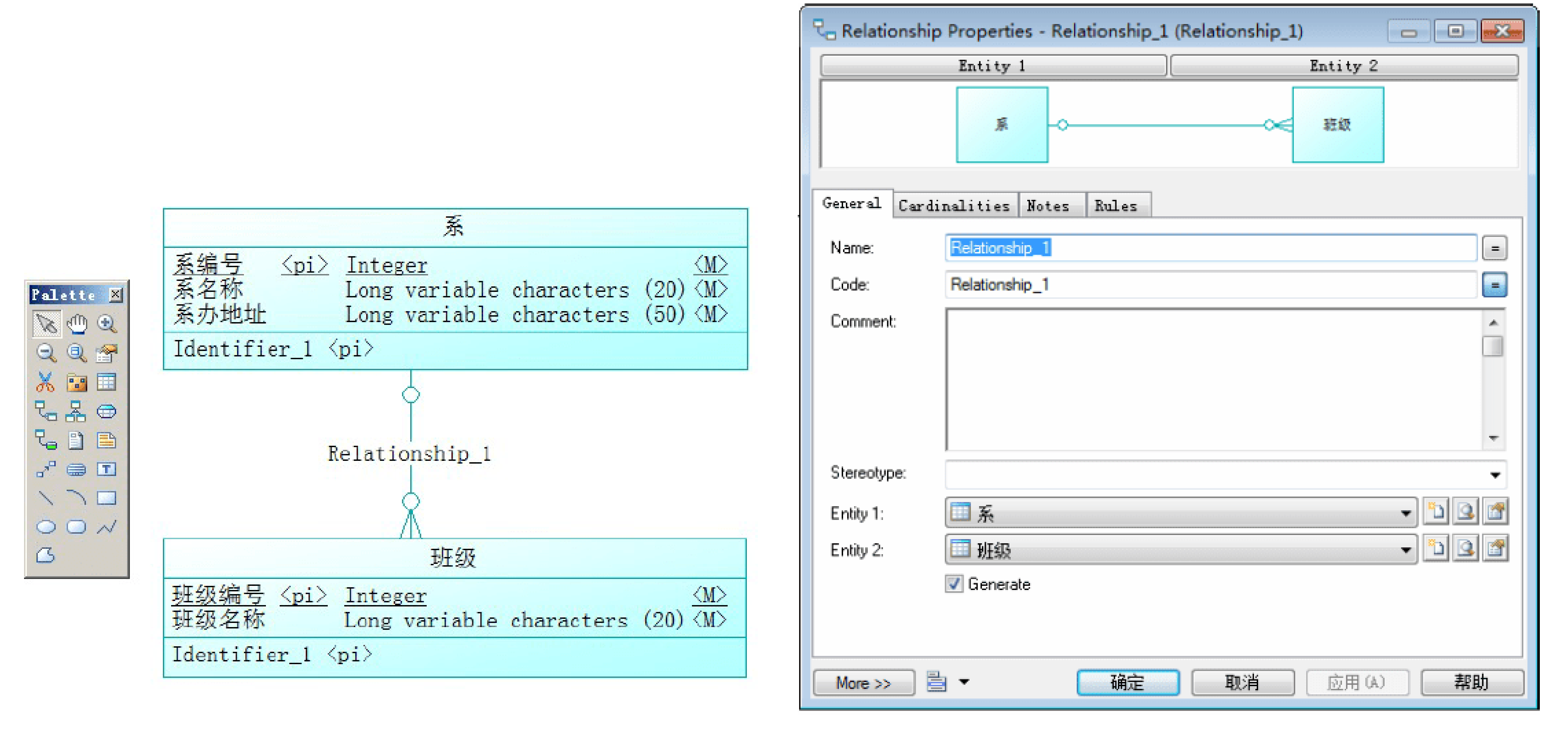


图 8.20　建立实体间的关系　　　　图 8.21　关系的属性对话框

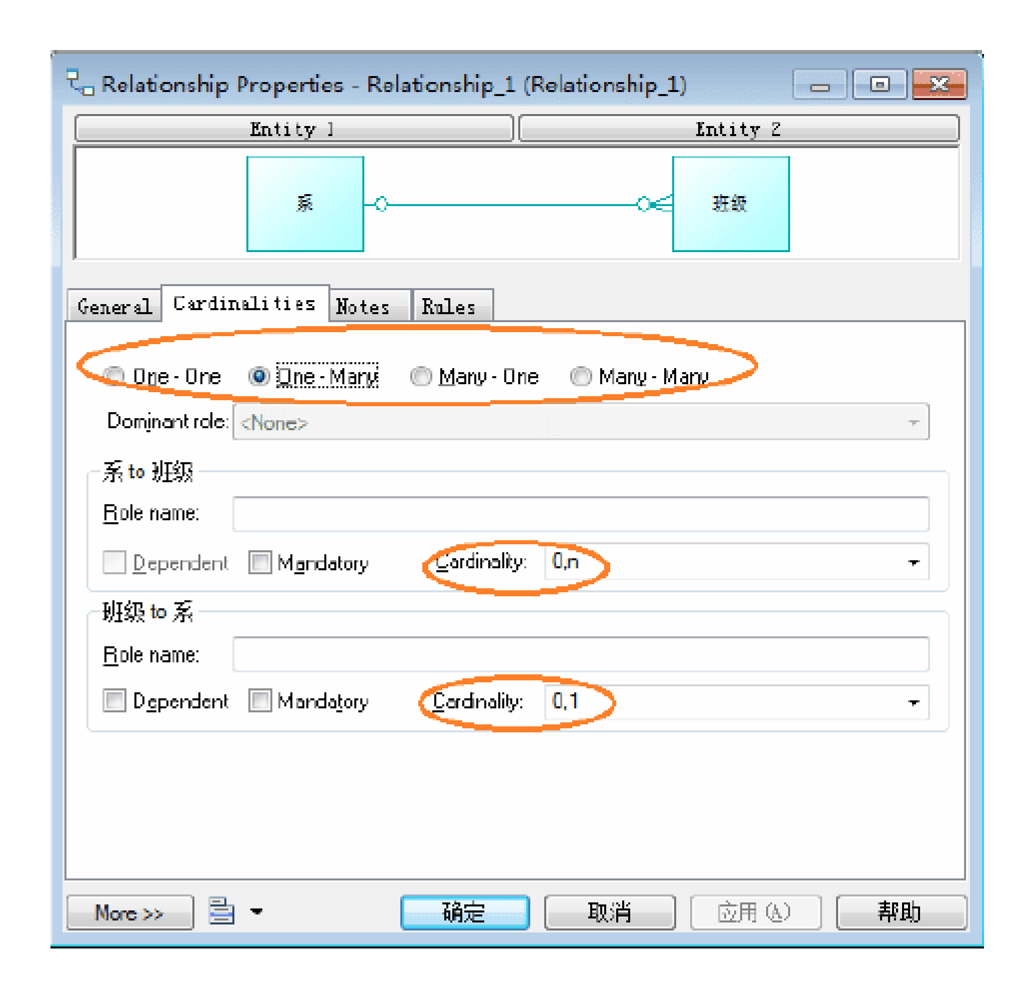


图 8.22　选择关系类型

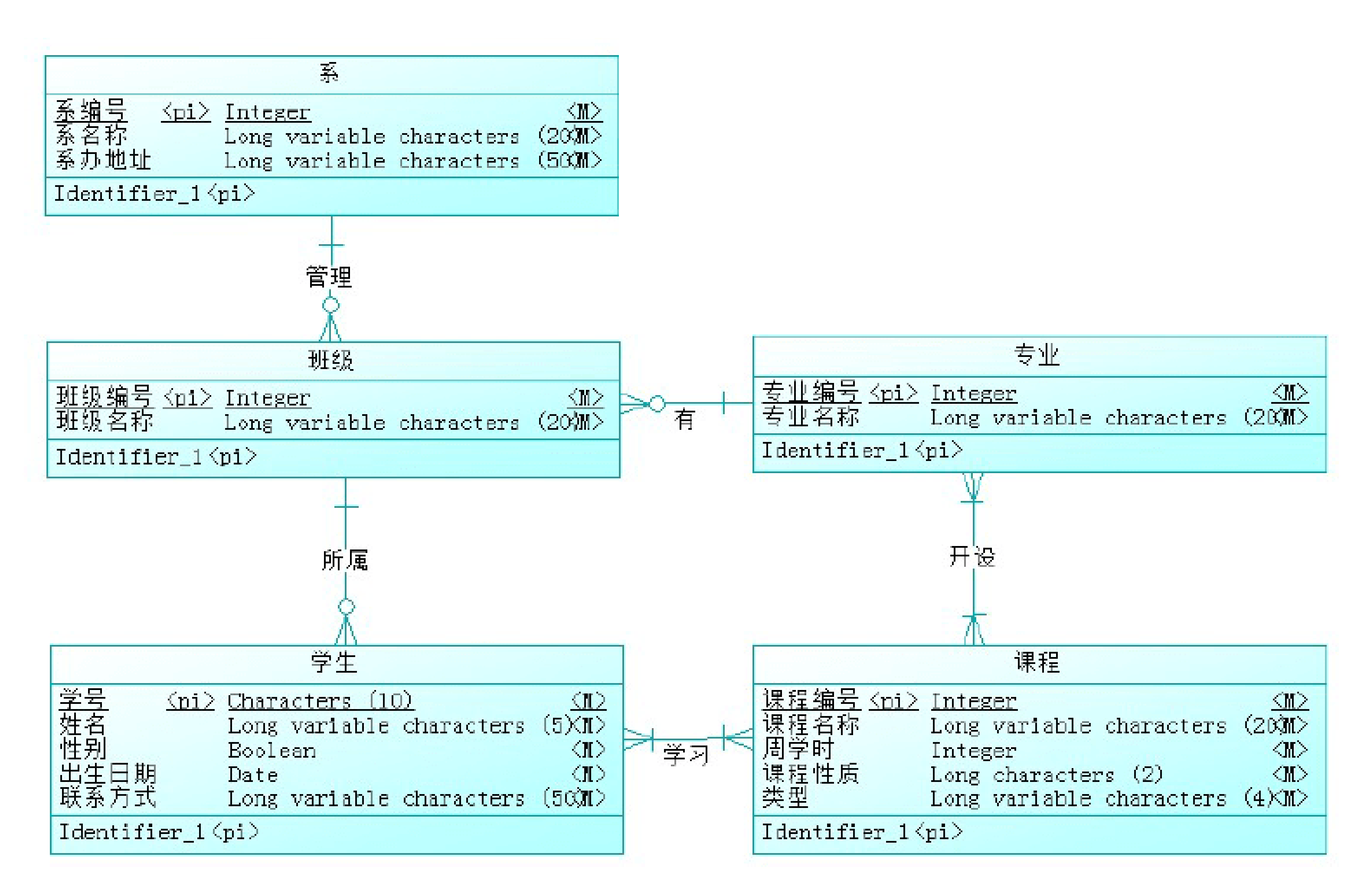


图 8.23　完整的概念模型

8.5.4　建立物理模型

物理模型是针对具体数据库实现的一种模型，由于 SQL Server 2012 刚刚推出，在目前最新版本的 PowerDesigner 15.1 中虽然支持 SQL Server 2012，但是并没有在列表中列出，后面的逆向工程使用时就会见 SQL Server 2012 的应用了。这里，先使用 SQL Server 2008，但是对于数据库建模不会有影响。

物理模型中表现了表与表的关系，PowerDesigner 支持从概念模型转换为物理模型。简单地讲，转换的过程就是将实体转换为表，关系转换为中间表和外键约束。在 PowerDesinger 中建立物理模型的操作如下所述。

（1）在 PowerDesigner 中打开概念模型，然后选择 Tools 菜单下的 Generate Physical Data Model 选项，系统将弹出产生物理模型选项，如图 8.24 所示。

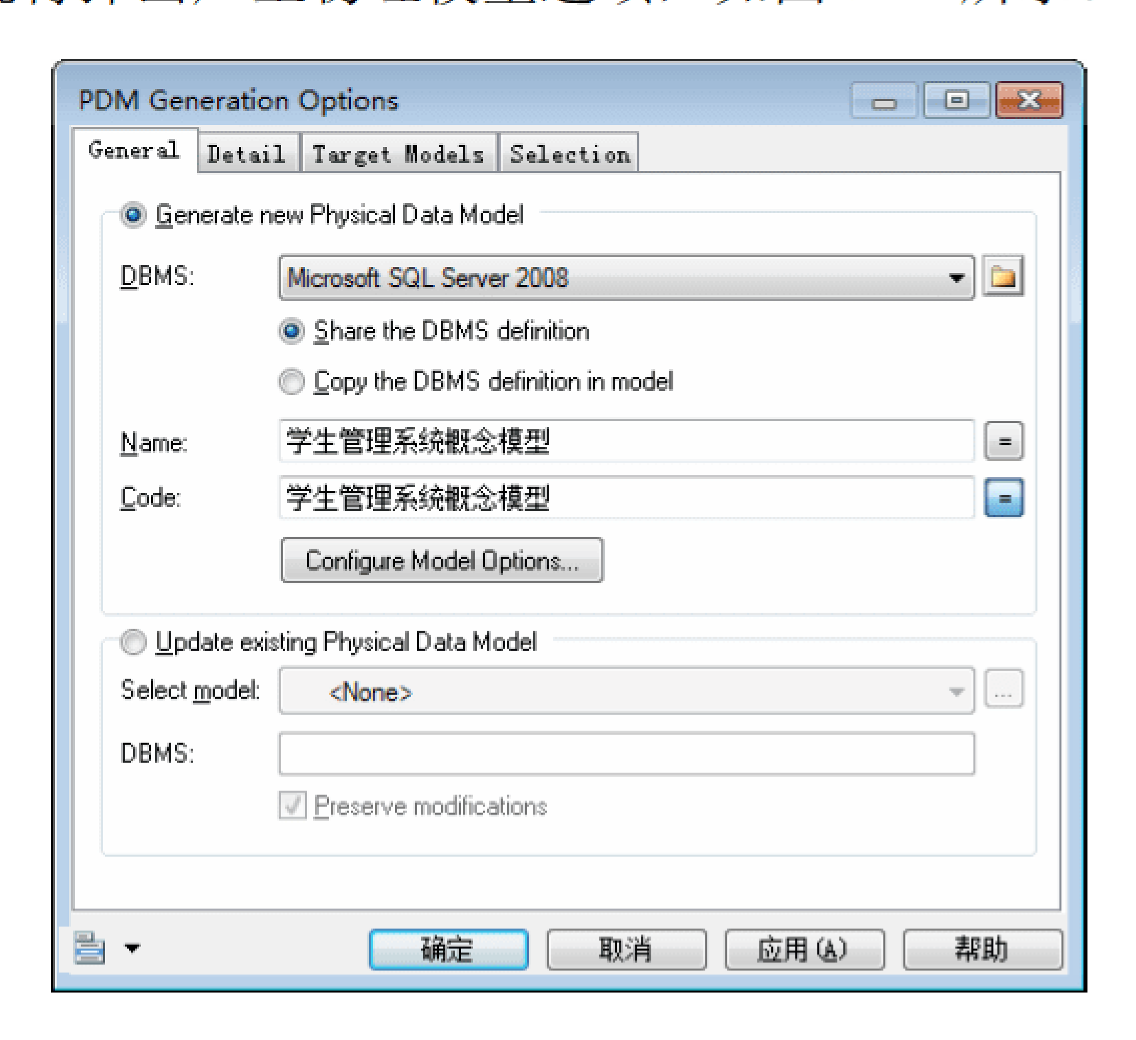


图 8.24　产生物理模型选项

（2）在 Name 文本框中输入物理模型的名字，然后单击“确定”按钮，系统将根据概念模型生成物理模型，生成的物理模型如图 8.25 所示。

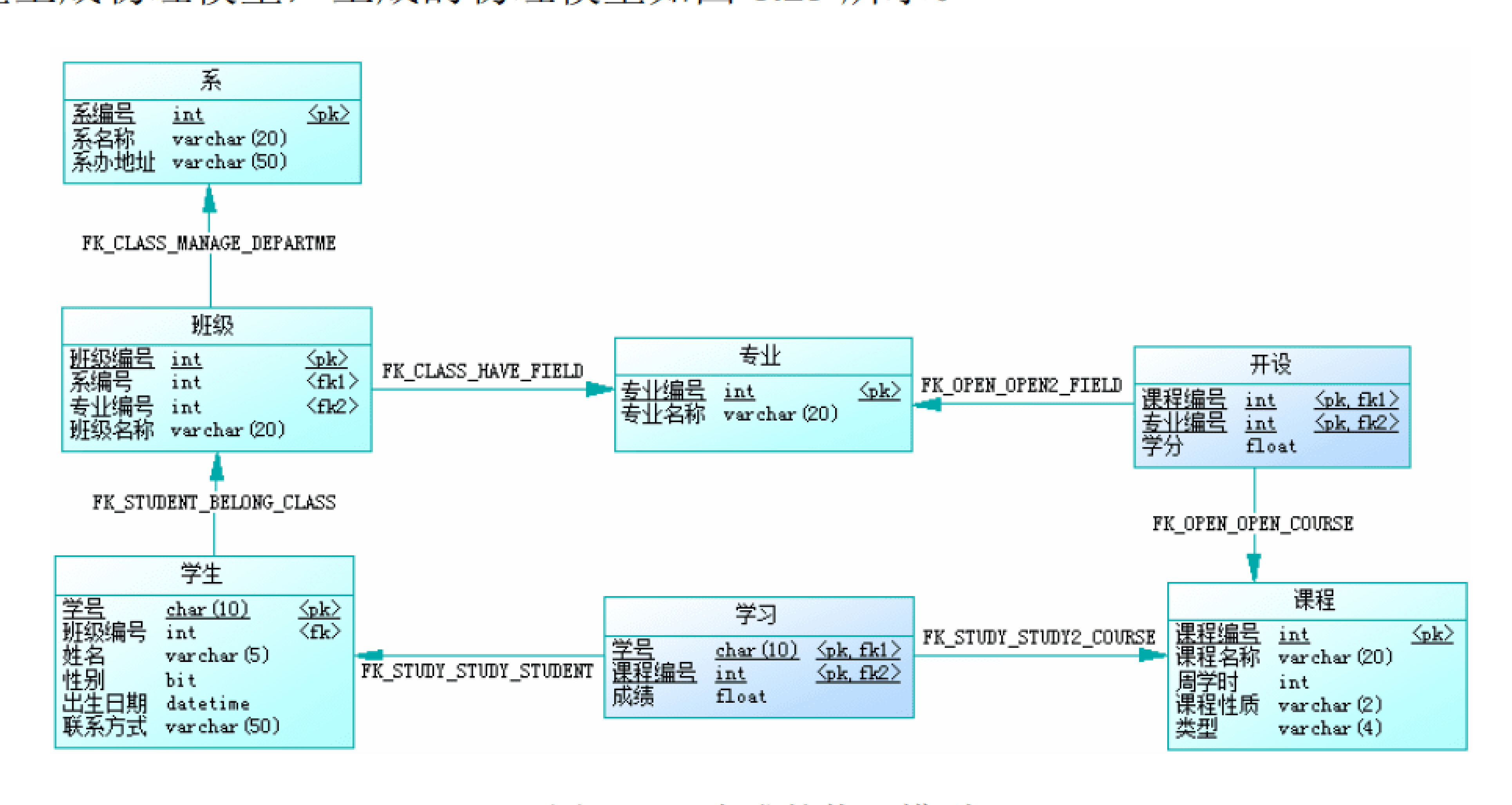


图 8.25　生成的物理模型

从生成的物理模型图中可以看到，多对多关系已经被中间表代替，而一对多关系也转换为外键约束，同时将“多”侧表的主键添加到“一”侧中作为其中的列。例如班级表中，系统添加了系编号和专业编号列。

（3）在主设计面板中双击班级表，系统将弹出表属性窗口，选择 Columns 标签，切换到列设置界面，如图 8.26 所示。

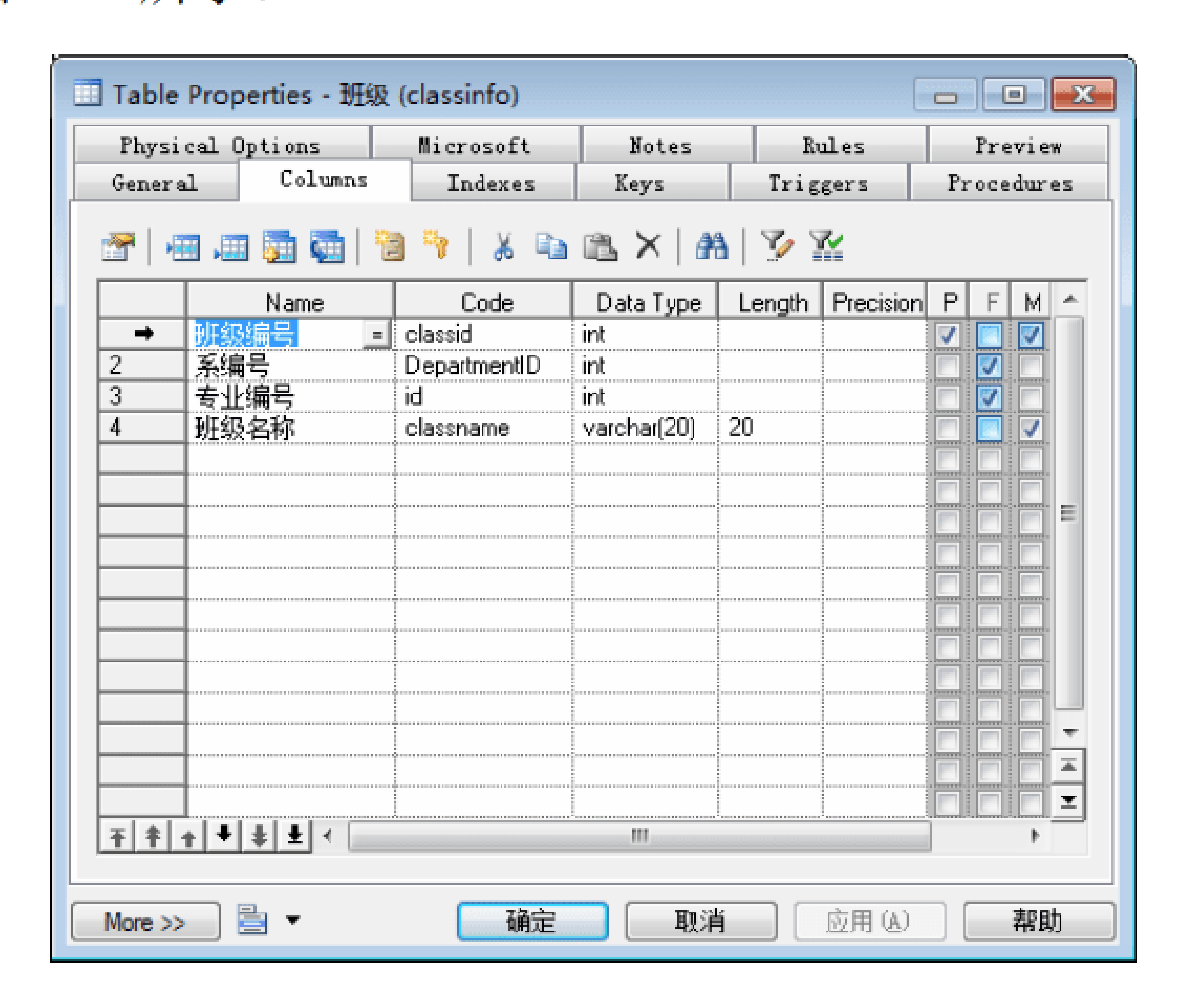


图 8.26　表的列设置

（4）在选中“班级编号”的情况下，单击工具栏的 Properties 按钮，或者使用快捷键 Alt＋Enter，系统将打开该列的属性对话框，如图 8.27 所示。

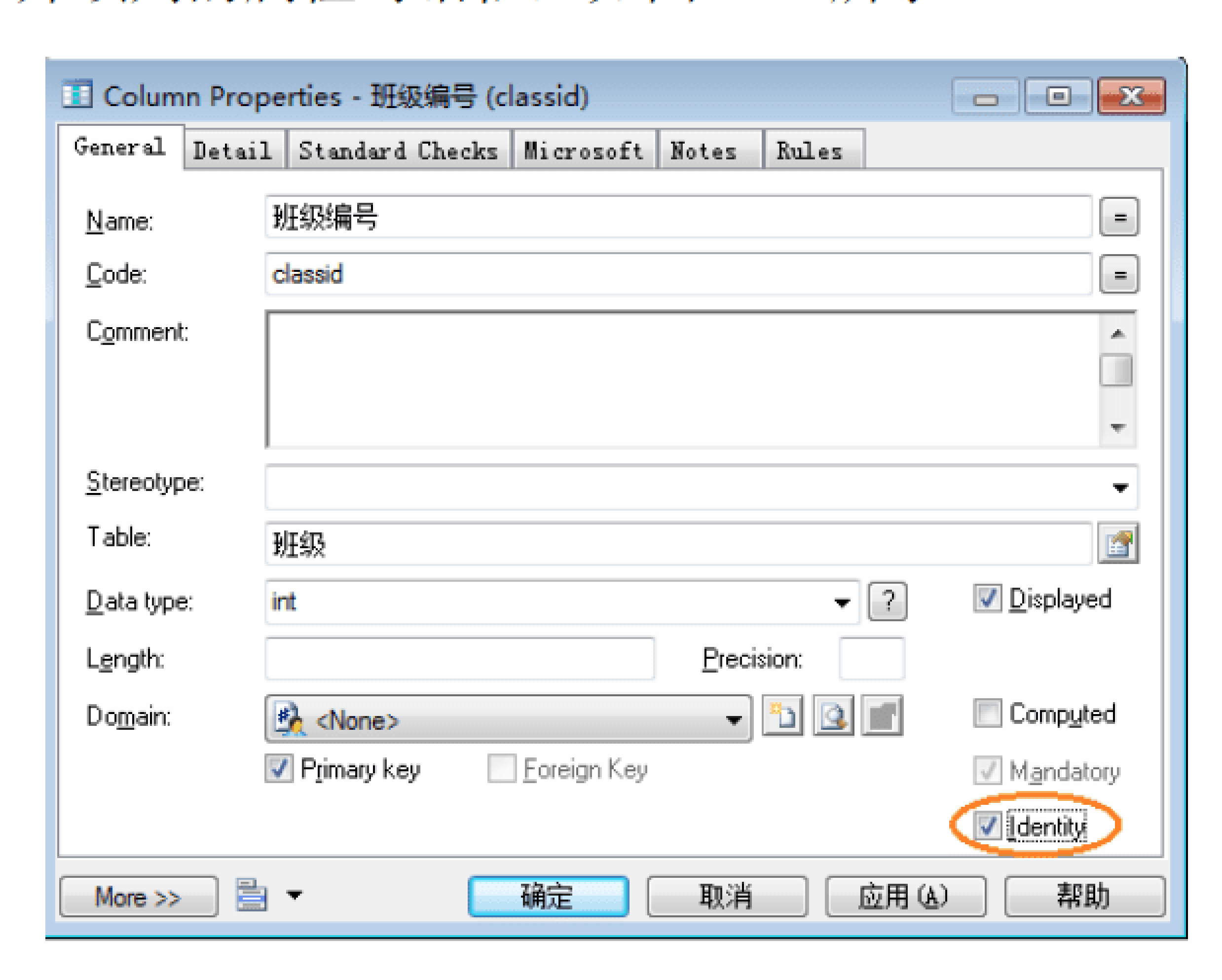


图 8.27　列属性对话框

（5）由于该列是主键，在数据库中如果希望将该列设置为自增列，可以选中 Identity 复选框，然后单击“确定”按钮回到表属性对话框。

（6）单击“确定”按钮回到主设计界面，用同样的方法设置需要使用自增列的表。

（7）双击“学习”表，打开该表的属性窗口，选择 Columns 标签，切换到列设置界面。

（8）添加“成绩”列，数据类型为 float，并且允许为空。

（9）单击“确定”按钮完成成绩列的添加。用同样的方式为“开设”表添加“学分”列，数据类型为 float，并且不能为空。

至此整个物理模型也设置完成了。在整个物理模型中完整地表示了 E-R 图中的所有信息。

8.5.5　生成数据库

PowerDesigner 支持从数据库模型到数据库表的建立，同样也可以根据现有数据库生成物理模型，这就是 PowerDesigner 的正向工程和逆向工程。

- 正向工程：能直接从 PDM 中产生一个数据库或产生一个能在用户的数据库管理系统环境中运行的数据库脚本。可以生成数据库脚本，如果选择 ODBC 方式，则可以直接连接到数据库，从而直接产生数据库表以及其他数据库对象。
- 逆向工程：根据已存在的数据库产生 PDM。数据来源可能是一个脚本文件，也可以是一个数据库连接。

首先讲正向工程——通过物理模型建立数据库。建立数据库的最好方式是使用 PowerDesigner 生成数据库脚本，然后将脚本复制到 SSMS 中运行。笔者不建议使用 ODBC 的方式直接连接到数据库修改表，因为这将使数据库的变动不易控制。在 PowerDesigner 中生成数据库脚本的操作如下所述。

（1）使用 PowerDesigner 打开物理模型。

（2）选择 Database|Generate Database 命令，系统将打开数据库生成配置对话框，如图 8.28 所示。

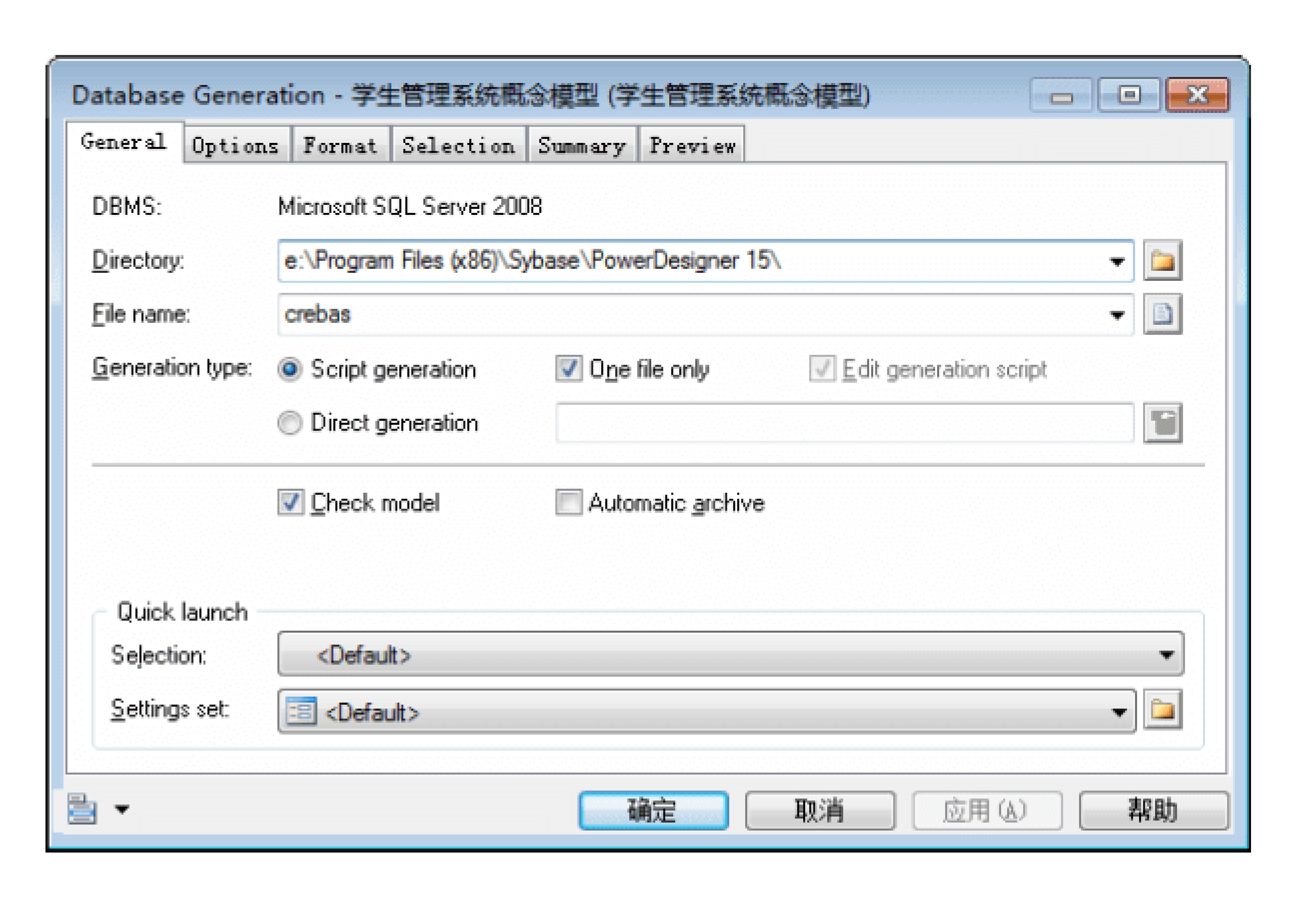


图 8.28　数据库生成配置对话框

（3）选择将生成的脚本文件保存到本地硬盘的哪个位置，输入脚本文件的文件名。选中 Check model 复选框，让系统在生成脚本前检查模型，看是否有不符合规范的地方。

（4）选择 Format 标签，切换到格式化设置界面。对于 T-SQL 中的一些标识关键字，若出现在了模型中，则应该使用“[]”括起来。另外，为了脚本清晰易懂，应该选中 Generate name in empty comment 复选框，将列的名字作为 SQL Server 数据库中列的注释生成到脚本文件中，配置如图 8.29 所示。

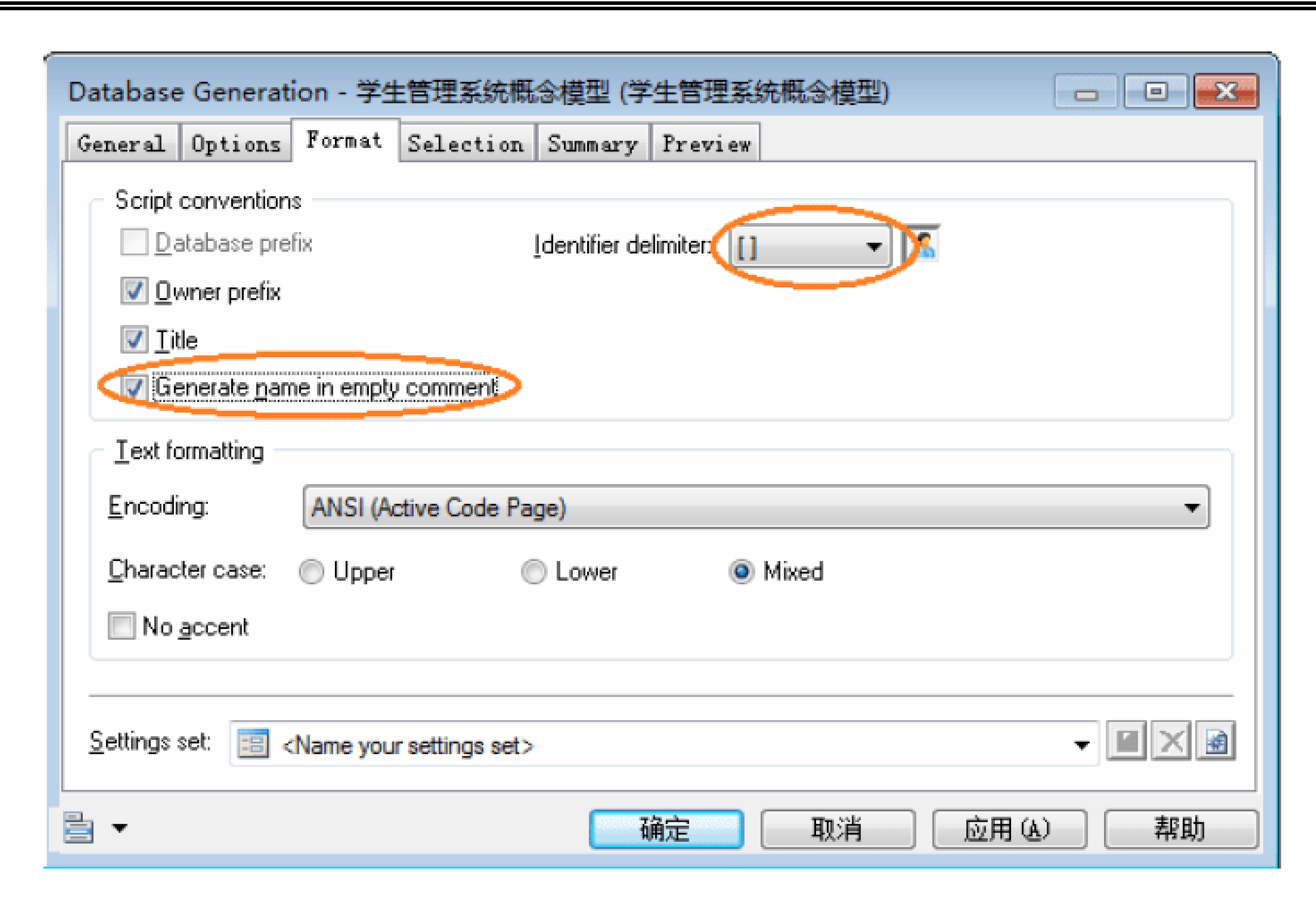


图 8.29　格式化设置

（5）单击“确定”按钮，系统将根据物理模型和设定生成数据库脚本文件到指定位置。

（6）将生成的脚本文件内容复制到 SSMS 中，选定运行的数据库运行即可创建物理模型对应的数据库对象。

注意： 这里生成的脚本内容不包含创建数据库，所以需要先手动创建数据库，再在新建的数据库中运行脚本。

8.5.6　使用逆向工程生成物理模型

对于已有的数据库，若需要将其生成物理模型，则需要使用 PowerDesigner 中的逆向工程。在 PowerDesigner 中进行逆向工程生成物理模型的操作步骤如下所述。

（1）选择 File|Reverse Engineer|Database 命令，系统将弹出逆向工程选择对话框，如图 8.30 所示。

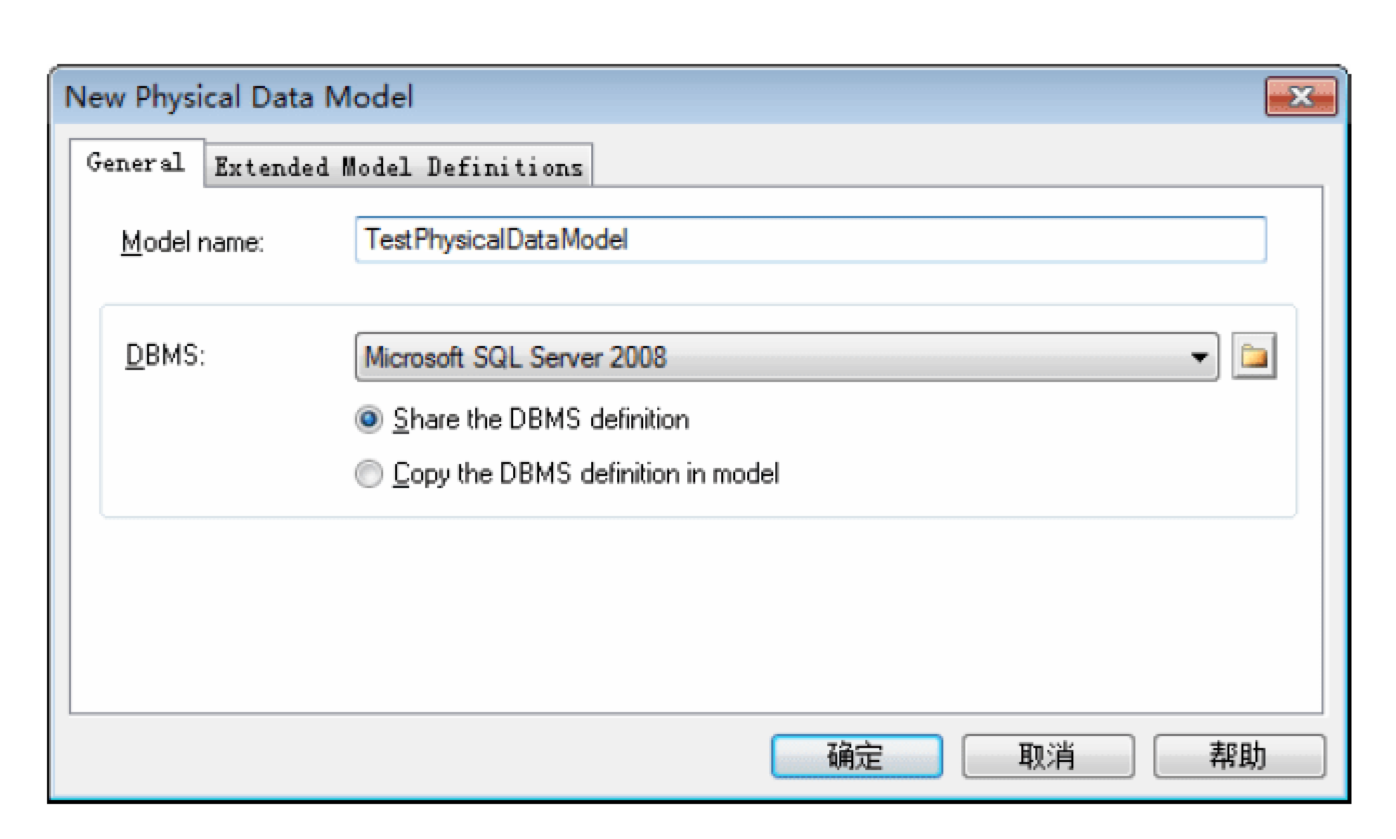


图 8.30　逆向工程新建物理模型对话框

（2）输入要新建的物理模型的名字，在 DBMS 下拉列表框中选择 SQL Server 2008 数据库，然后单击“确定”按钮，系统将进入数据库逆向工程引擎设置对话框。如图 8.31 所示。

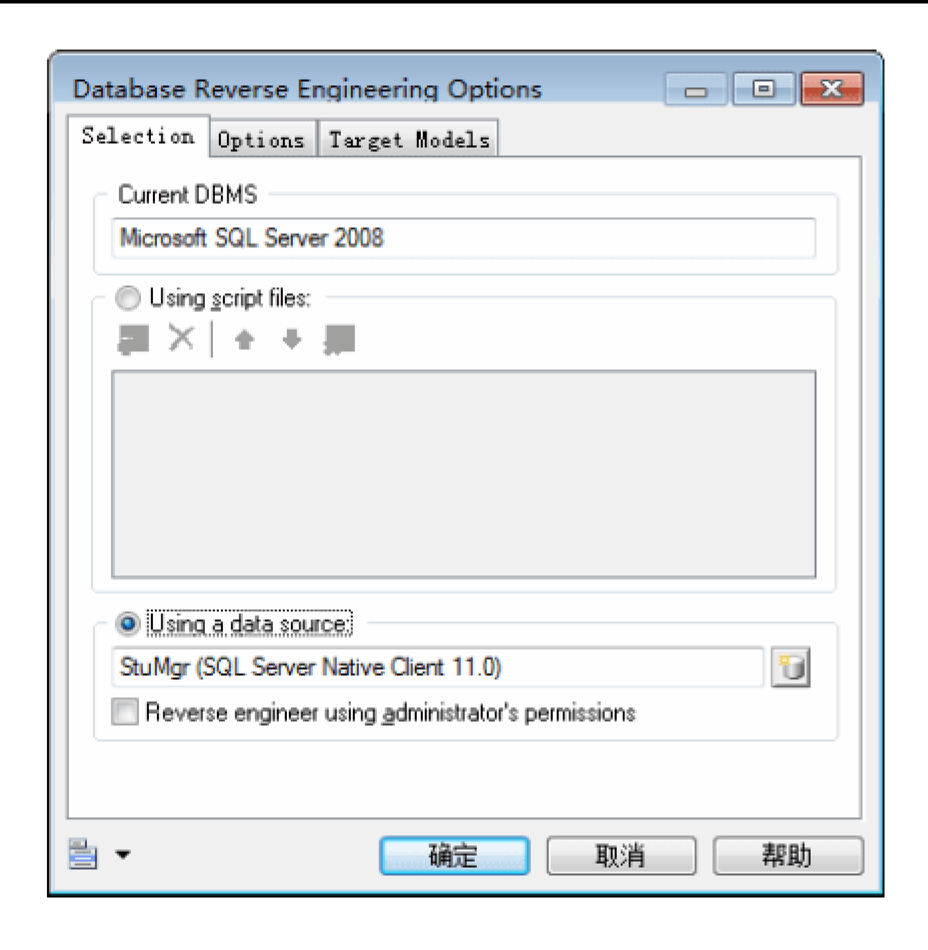


图 8.31　数据库逆向工程引擎设置

（3）由于这里是使用连接到数据库利用数据库中的表建立物理模型，所以选择 Using a data source 单选按钮。然后单击 Connect to a data source 按钮，系统将弹出数据源设置对话框，如图 8.32 所示。

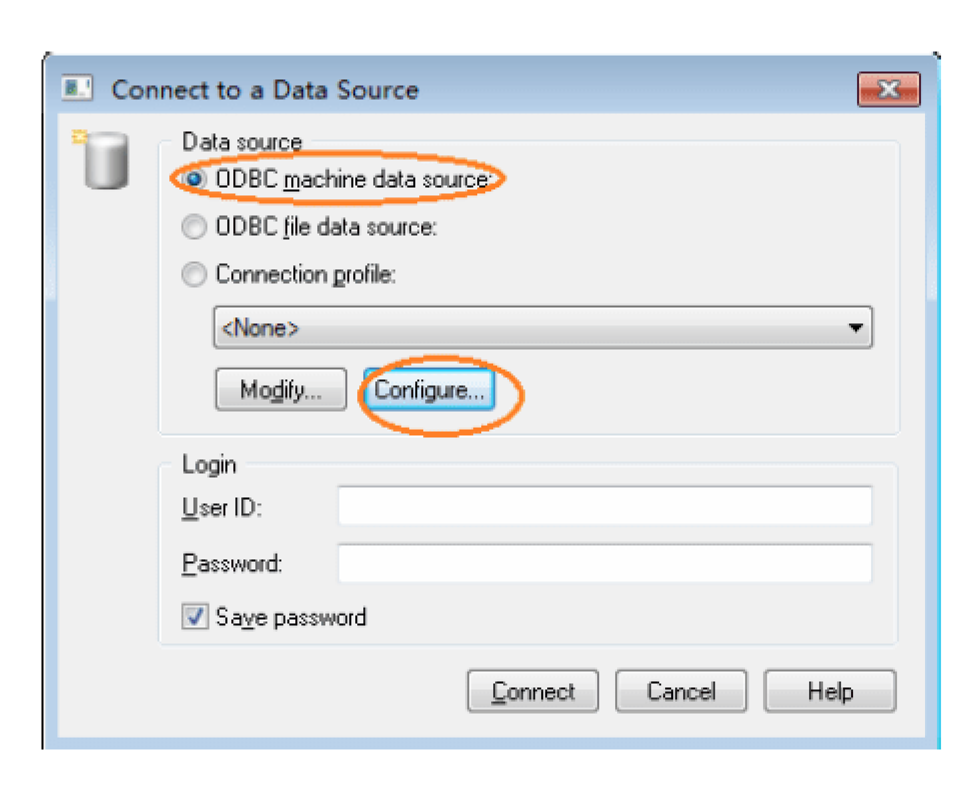


图 8.32　数据源设置

（4）选择 ODBC machine data source 单选按钮，然后单击 Configure 按钮，系统将弹出数据连接配置对话框，如图 8.33 所示。

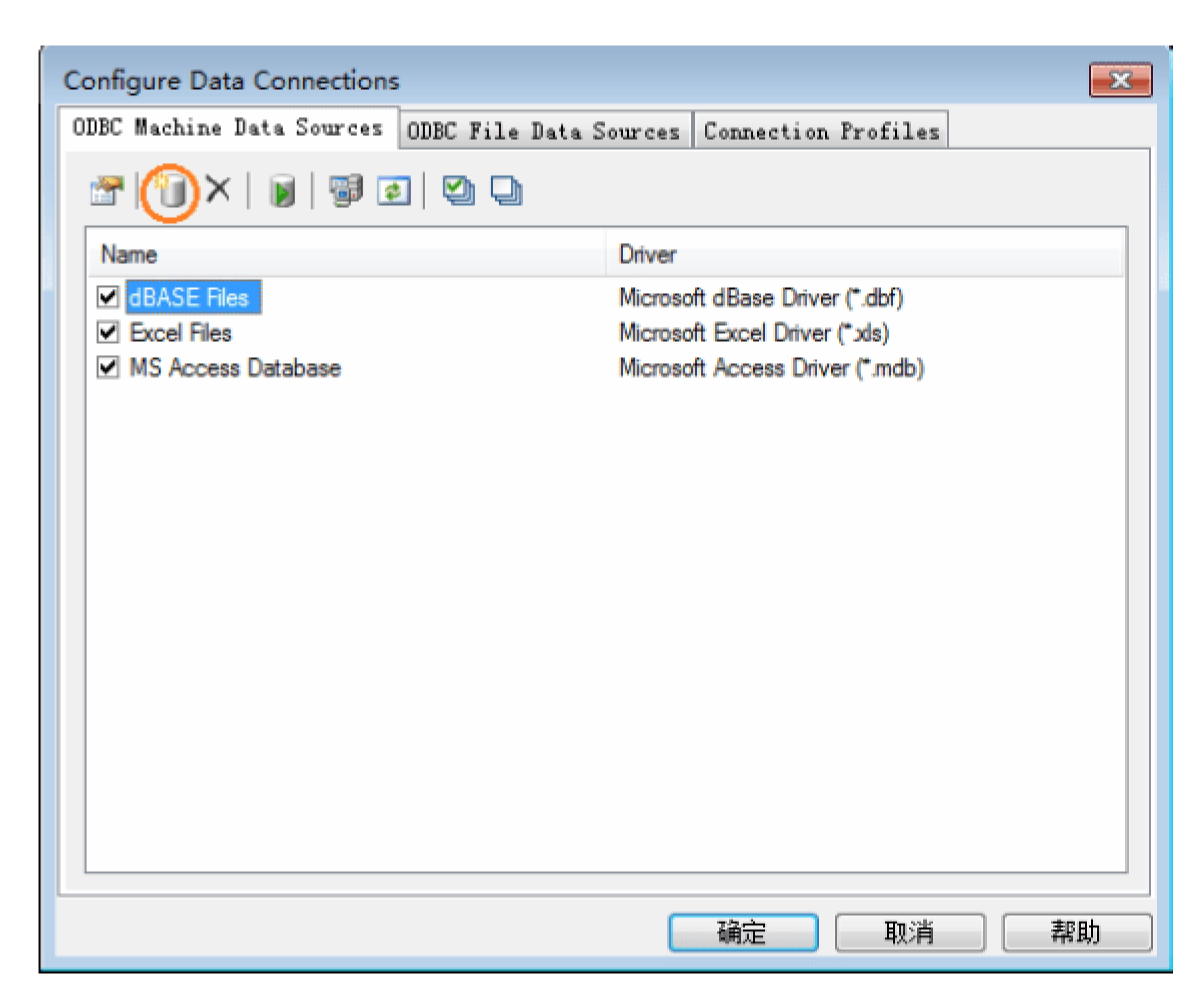


图 8.33　数据连接配置

（5）单击工具栏的 Add Data Source 按钮添加一个新的数据库连接，系统弹出“创建新数据源”对话框，如图 8.34 所示。

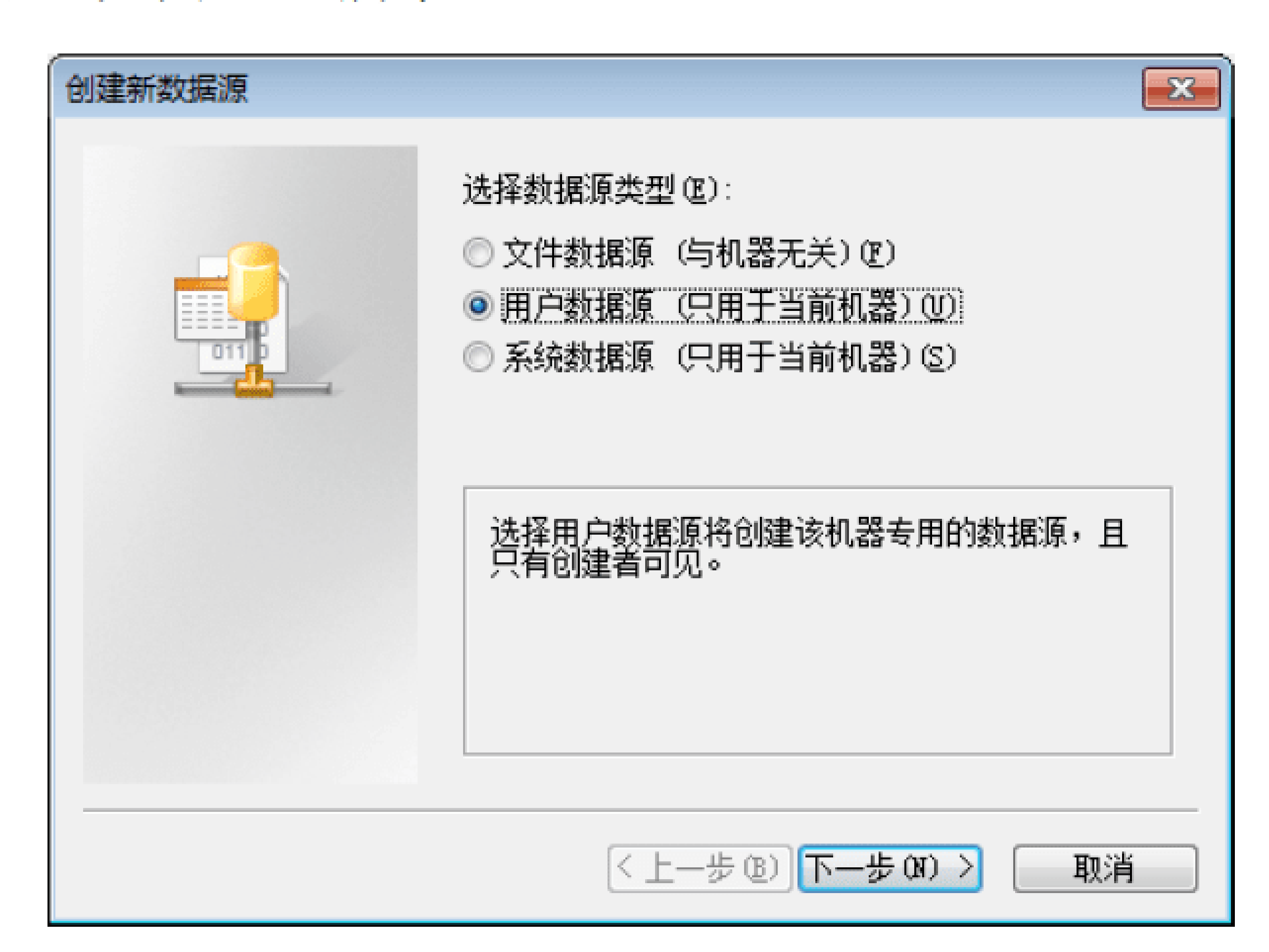


图 8.34　“创建新数据源”对话框

（6）选择“用户数据源（只用于当前机器）”单选按钮，然后单击“下一步”按钮，向导进入驱动程序选择界面，如图 8.35 所示。

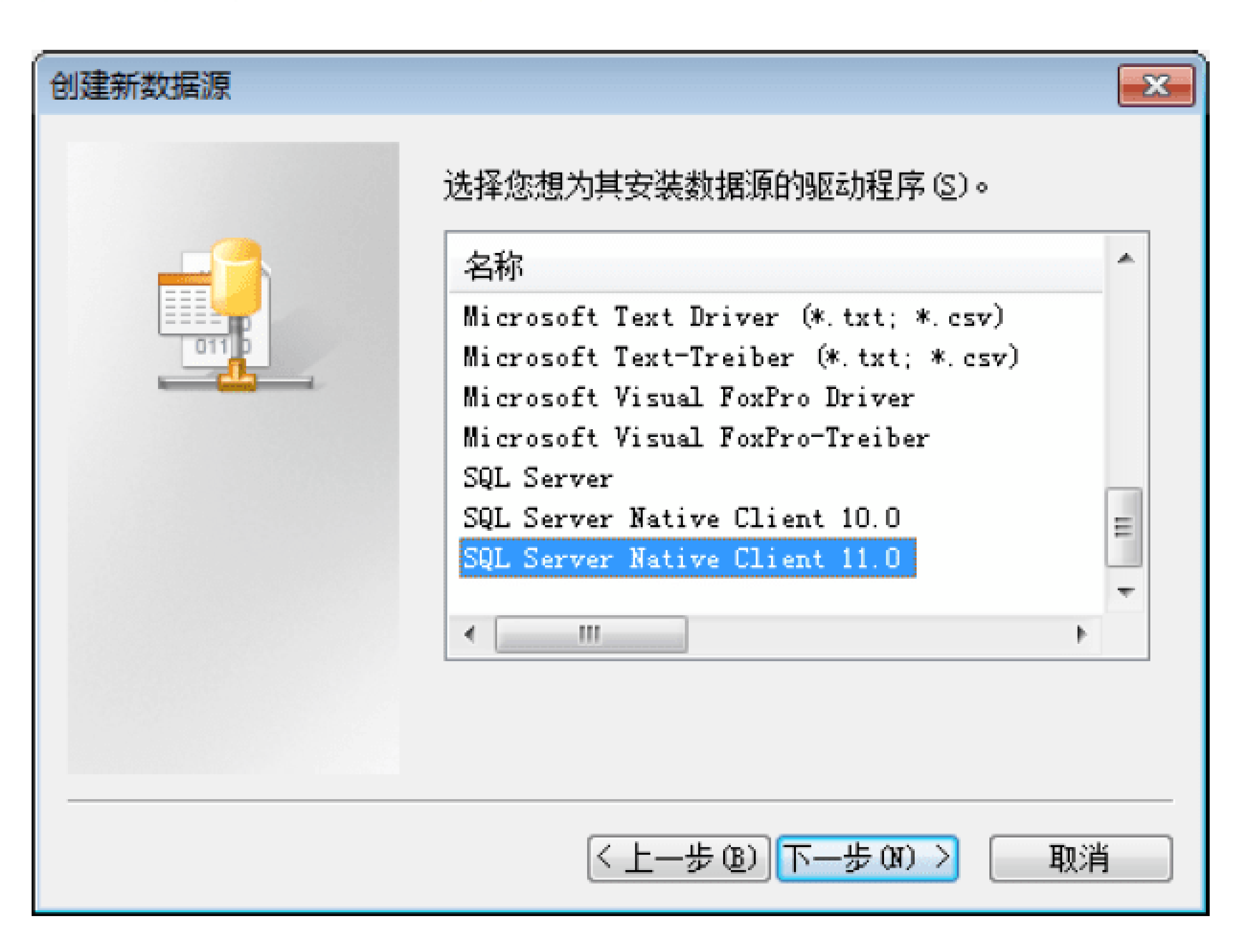


图 8.35　数据库驱动选择

注意：数据源驱动程序列表中列出的是当前 PowerDesigner 所在计算机的驱动，如果没有安装 SQL Server 2012，则不会有 SQL Server Native Client 11.0 选项。

（7）由于这里是要连接 SQL Server 2012 数据库，所以选中该数据库提供的驱动 SQL Server Native Client 11.0 选项。单击“下一步”按钮，系统提示当前选择的驱动，然后单击“完成”按钮，系统将调用 SQL Server 2012 创建数据源的向导，如图 8.36 所示。

（8）输入要新建的数据源的名称、描述和连接到的服务器 IP 地址或名称，这里，由于笔者将 SQL Server 2012 与 PowerDesigner 安装在同一台计算机上，所以服务器列表中是连接本地。单击“下一步”按钮进入用户设置界面，如图 8.37 所示。

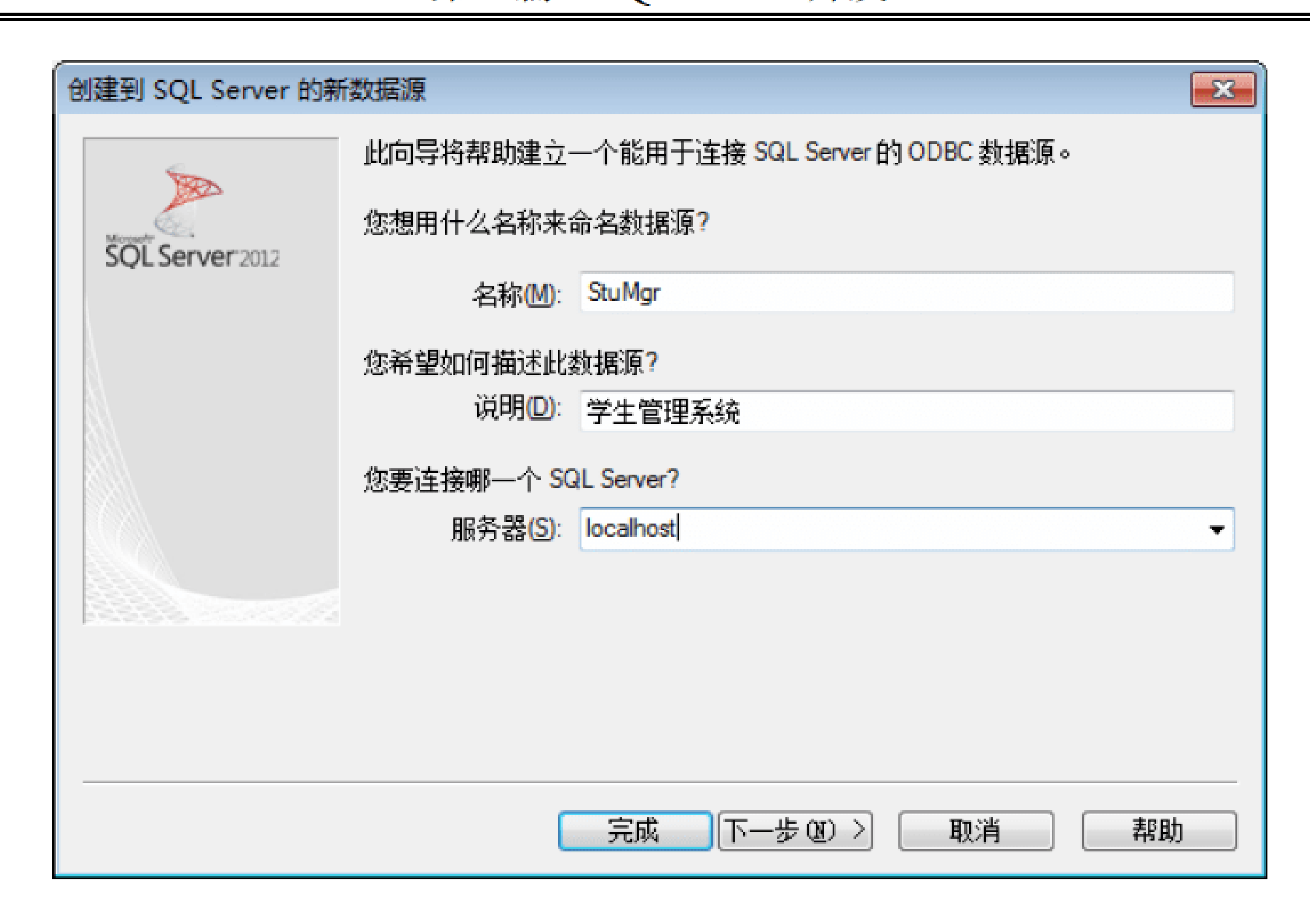


图 8.36　创建到 SQL Server 的新数据源



图 8.37　连接 SQL Server 的用户设置

（9）由于这里笔者连接的是本地 SQL Server，所以选择“集成 Windows 身份验证”单选按钮。单击“下一步”按钮进入默认数据库设置界面，如图 8.38 所示。

（10）选中“更改默认的数据库为”复选框，然后在数据库下拉列表框中选择要进行逆向工程的数据库 StuMgr。单击“下一步”按钮，系统进入数据库其他设置界面，如图 8.39 所示。

（11）这里根据实际情况来选择和更改设置选项，一般保持默认值即可。修改后单击“完成”按钮，完成了创建 SQL Server 数据源的操作。系统将把向导中所有配置信息再次展示出来，如果确认无误，单击“确定”按钮，SQL Server 数据源创建完成，系统回到数据连接配置对话框。

（12）从数据连接配置对话框中可以看到新建的 StuMgr 数据源已经在其中了。单击“确定”按钮回到连接到一个数据源窗口，如图 8.40 所示。

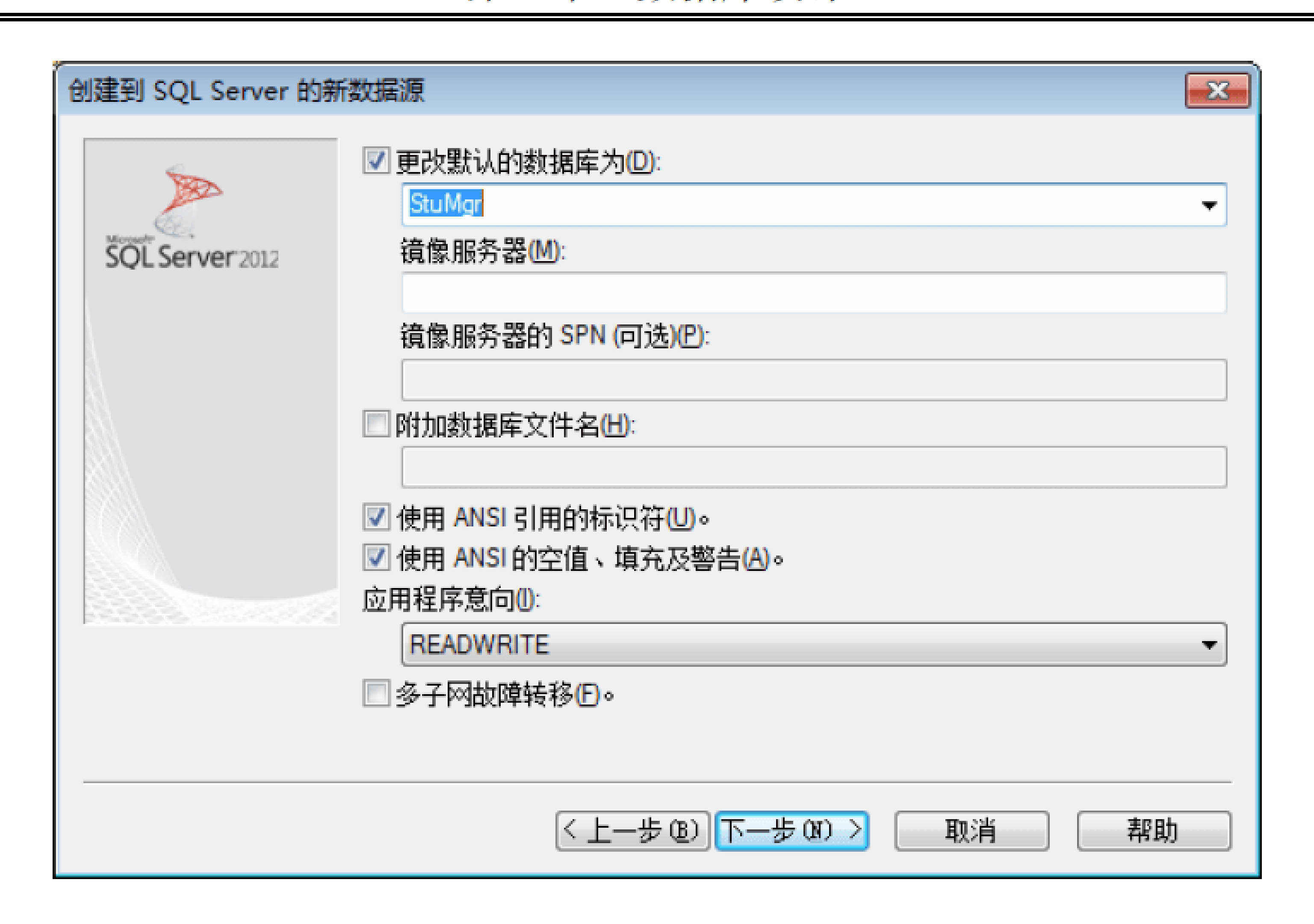


图 8.38　默认数据库设置

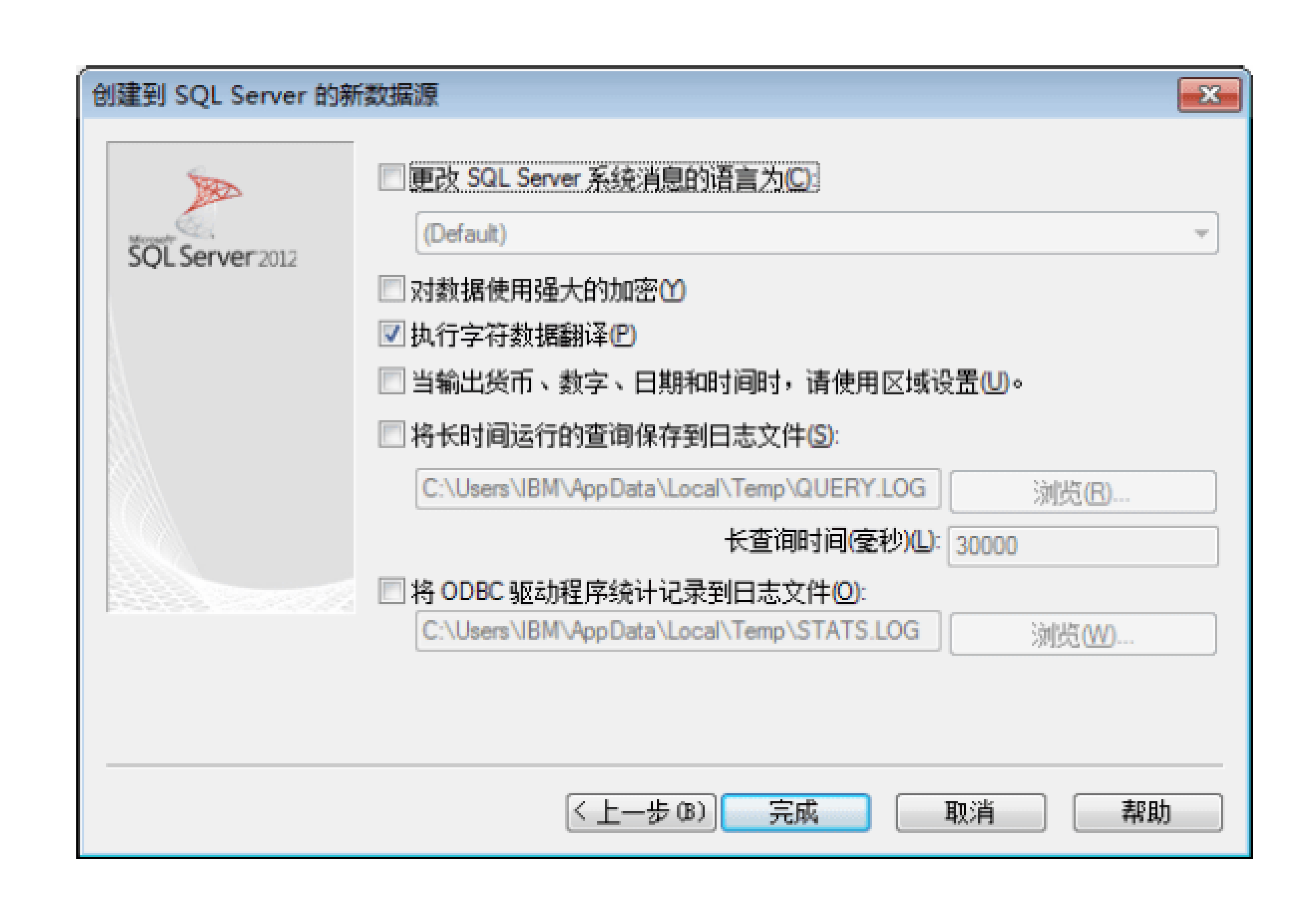


图 8.39　数据库其他设置

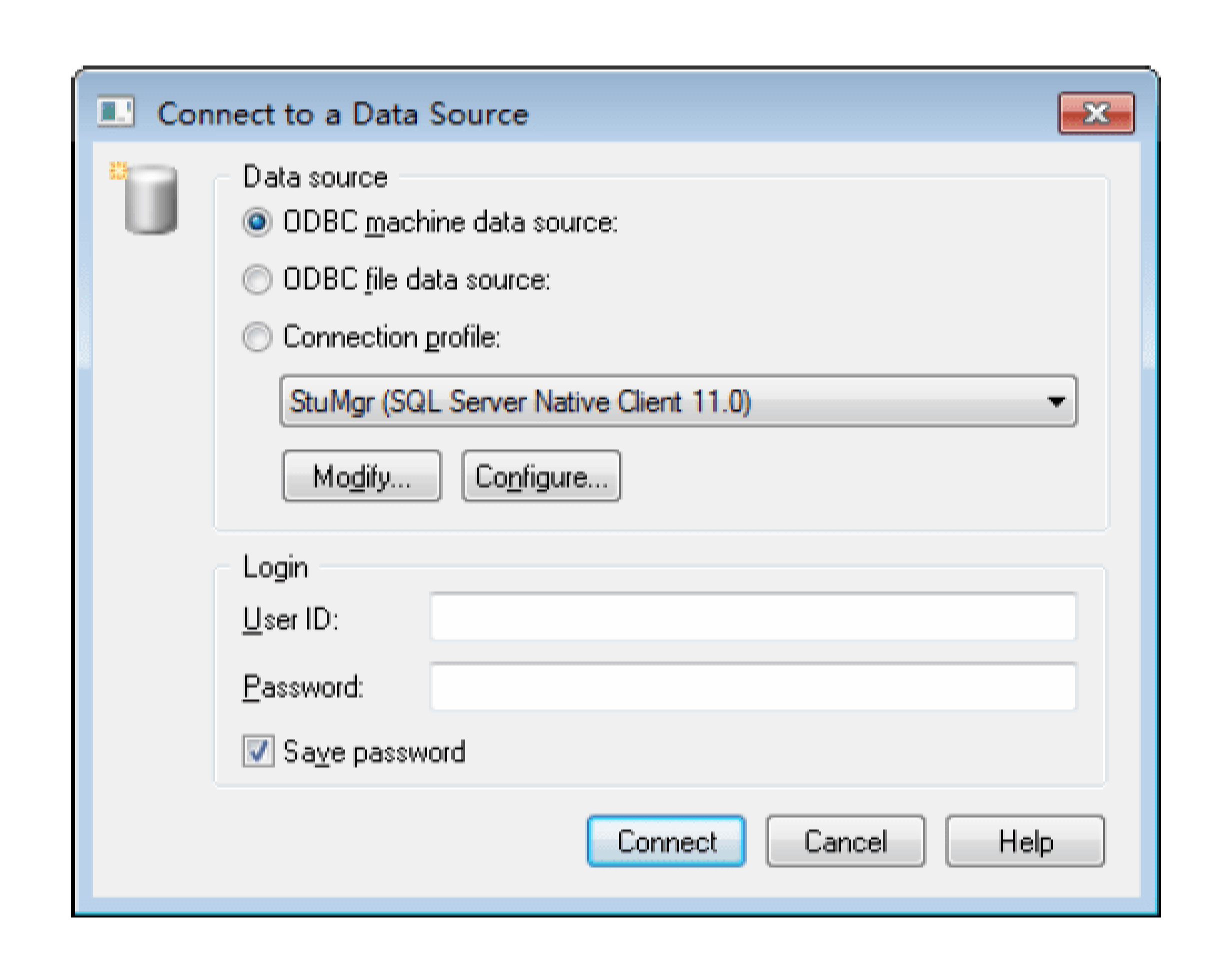


图 8.40　连接到一个数据源

（13）在数据源下拉列表框中选择 StuMgr 数据源，由于这里使用的是 Windows 身份认证进行连接，所以不需要输入用户名和密码，直接单击 Connect 按钮即可，系统回到 PowerDesigner 的数据库逆向工程选项对话框。

（14）单击“确定”按钮，系统将进入数据库对象选择界面，列出数据源中的所有数据库对象和需要进行逆向工程的其他对象选项。

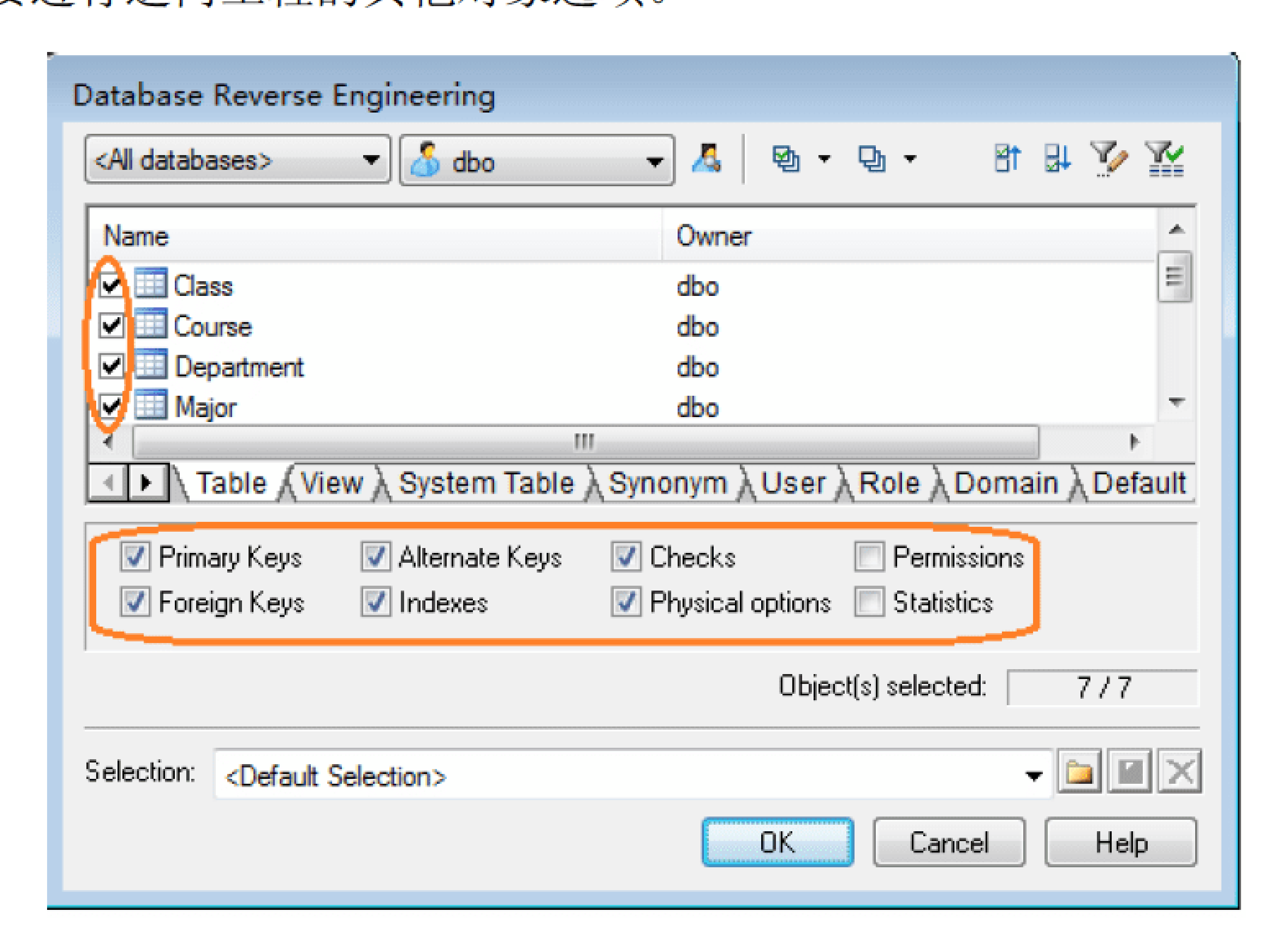


图 8.41　需要逆向工程的数据库对象设置

（15）选择需要逆向工程的表，至少保证主键和外键是要被逆向工程的，其他索引和约束等则根据实际情况确定是否需要进行逆向工程。选择好需要逆向工程的表后单击 OK 按钮，系统将分析数据库系统中的数据库对象并生成物理模型。如图 8.42 所示为根据 StuMgr 数据库生成的物理模型。

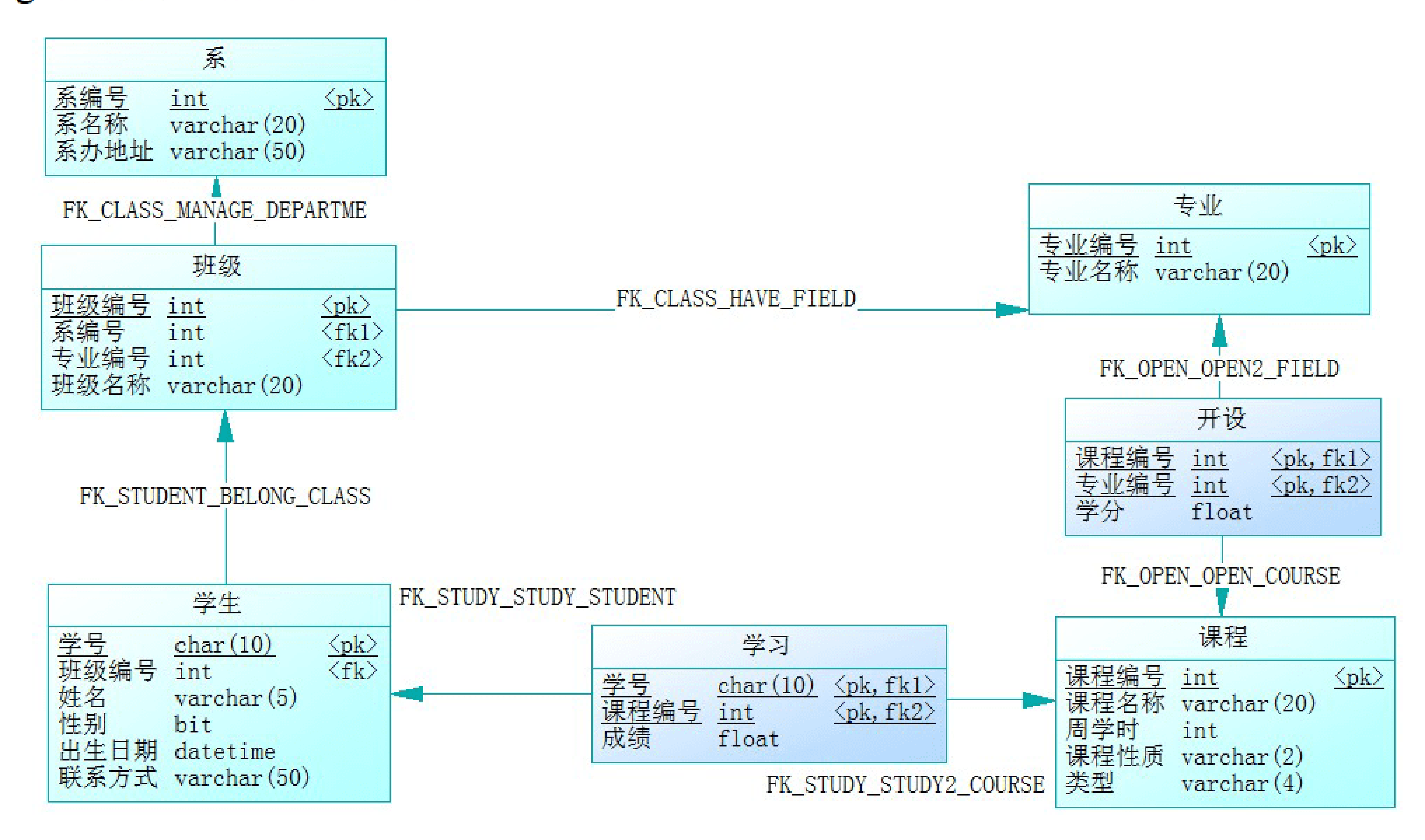


图 8.42　逆向工程生成的物理模型

除了通过数据库生成物理模型外，还可以将物理模型生成概念模型。由物理模型生成概念模型与由概念模型生成物理模型的操作相似，选择 Tools 菜单下的 Generate Conceptual Data Model 选项即可。

8.6 小　　结

数据库设计是一个很大的概念，很多设计方法和技巧都需要通过大量的实践积累才能用到。许多优秀的书都将数据库设计作为核心内容进行讲解，其博大精深不可能在这一章中就能讲解完的。本章主要从关系数据库的基本概念出发讲解了实体—关系模型、实体之间的关系和数据库设置中的范式，最后侧重讲解了使用 PowerDesigner 进行数据库建模的方法。

数据库设计中的表一般只需要满足第三范式即可，能够满足 BCNF 范式规则已是很好的数据库设计。在数据库设计中最基本也最重要的工具就是 E-R 图。首先需要分析设计出合理的 E-R 图，然后使用 PowerDesigner 等建模工具将 E-R 图转换为概念模型、物理模型并最终转换为数据库中的对象。PowerDesigner 除用于数据库建模外还提供了 UML 建模等功能。

第 9 章　SQL Server 与 CLR 集成

SQL Server 从 2005 版起就支持使用 CLR 来编写数据类型、存储过程、函数、触发器和聚合等。SQL Server 与 CLR 的集成显著地增强了其数据库编程模型的功能，扩展了 T-SQL 的处理能力。本章将全部使用 C#语言进行 CLR 开发，面向的是对 C#或者其他.NET 语言有一定基础的读者。读者若不懂 C#的话可以跳过本章或补充用 C#进行编程的基本知识。本章将主要讲解如何在 SQL Server 中使用 CLR 扩展存储过程、函数、聚合、数据类型和触发器。

9.1　了解.NET 和 CLR

要了解 CLR 必须要知道微软的.NET 平台及其代表语言 C#，对这些基础知识有所了解有助于对本章内容的学习和理解。

9.1.1　.NET 简介

对于 Microsoft .NET，微软官方有如下描述：

“.NET 是 Microsoft 的用以创建 XML Web 服务（下一代软件）的平台，该平台将信息、设备和人以一种统一的、个性化的方式联系起来。”

“借助于.NET 平台，可以创建和使用基于 XML 的应用程序、进程和 Web 站点以及服务，它们之间可以按设计、在任何平台或智能设备上共享和组合信息与功能，以向单位和个人提供定制好的解决方案。”

“.NET 是一个全面的产品家族，它建立在行业标准和 Internet 标准之上，提供开发（工具）、管理（服务器）、使用（构造块服务和智能客户端）以及 XML Web 服务体验（丰富的用户体验）。.NET 将成为您今天正在使用的 Microsoft 应用程序、工具和服务器的一部分，同时，新产品不断扩展 XML Web 的服务能力以满足您的所有业务需求。”

从以上的官方描述中可以看到.NET 平台中超前的思想和优秀的目标。以.NET 平台定义的目标——微软为开发应用程序而创建的一个富有革命性的新平台.NET Framework。

尽管.NET Framework 的 Microsoft 版本运行在 Windows 操作系统上，但以后将推出运行在其他操作系统上的版本，例如 Mono，它是.NET Framework 的开发源代码版本（包含一个 C#编译器），该版本可以运行在几个操作系统上，包括各种 Linux 版本和 Mac OS。许多这类项目正在开发，在读者阅读本书时可能就已发布了。另外，还可以在个人数字助手 PDA 类设备和一些智能手机上使用 Microsoft .NET Compact Framework（基本上是完整.NET Framework 的一个子集）。使用.NET Framework 的一个主要原因是它可以作为集

成各种操作系统的方式，实现了同一个程序，多平台运行。

.NET Framework 并没有限制应用程序的类型。使用.NET Framework 可以创建 Windows 应用程序、Web 应用程序、Web 服务和其他各种类型的应用程序。

.NET Framework 的设计方式保证它可以用于各种语言，包括接下来要介绍的 C#语言，以及 C++、Visual Basic、JScript，还有 Delphi、Python、Ruby 等语言也有对应的.NET 版本，目前还在不断推出更多的.NET 版本的语言。所有这些语言都可以访问.NET Framework，它们还可以彼此交互。C#开发人员可以使用 Visual Basic 程序员编写的代码，反之亦然。

.NET Framework 主要包含一个非常大的代码库，可以在客户语言（如 C#）中通过面向对象编程技术来使用框架中提供的功能。这个框架库分为不同的模块，以不同的命名空间存在于不同的 dll 文件中，这样就可以根据希望得到的结果来选择使用其中的各个部分。例如，一个模块包含 Windows 应用程序的构件，另一个模块包含联网的代码块，还有一个模块包含 Web 开发的代码块。一些模块还分为更具体的子模块，例如在 Web 开发模块中，有用于建立 Web 服务的子模块。

将框架分为不同模块目的是使不同的操作系统可以根据自己的特性，支持其中的部分或全部模块。例如，PDA 支持所有的核心.NET 功能，但不需要某些更深奥的模块。

在部分.NET Framework 库中定义了一些基本类型。类型是数据的一种表达方式，指定其中最基础的部分（例如 32 位带符号的整数和布尔类型等），这样以便使用.NET Framework 在各种语言之间进行交互操作，这种方式称为通用类型系统（Common Type System，CTS）。

除了支持这个库以外，.NET Framework 还包含.NET 公共语言运行库（Common Language Runtime，CLR），CLR 负责管理用.NET 库开发的所有应用程序的执行。

.NET Framework 从最初的 1.0 版开始经过几年的发展和积累，提供的功能和类库越来越庞大，目前已经发展到 4.5 版本。在 Windows 2003 之前的操作系统中都没有集成.NET Framework，用户若需要运行.NET 编写的程序则必须要安装.NET Framework。Windows Vista 和之后的 Windows 2008 都集成了.NET Framework，用户若使用这两款操作系统则不需要再安装。

注意：Windows 2003 中集成了.NET 1.1，而目前最新版本为 4.5，若需要运行.NET 2.0 及以后版本的.NET 程序，则需要安装对应的.NET Framework。

SQL Server 2012 在安装时会检测操作系统中是否已经安装了.NET Framework，若未安装则会自动将其安装。所以读者若将 SQL Server 2012 安装到本地计算机时不用担心是否安装了.NET Framework。

9.1.2　C#简介

C#是.NET 编程语言中最常用的一门语言，可以用于创建要运行在.NET CLR 上的应用程序，它从 C 和 C++语言演化而来，是 Microsoft 专门为使用.NET 平台而创建的。因为 C#是近几年才发展起来的，所以吸取了以前的不足，考虑了其他语言（如 C++、Java 等）的许多优点，并解决了这些语言的缺点。

使用 C#开发应用程序比使用 C++简单，因为其语法比较简单。但是，C#是一种强大的语言，在 C++中能完成的任务利用 C#也能完成。由于 C#是类型安全的语言，所以 C#

中与 C++比较高级的功能及等价的功能（例如直接访问和处理系统内存），只能在标记为“不安全”的代码中使用。因为它可能覆盖系统中重要的内存块，导致严重的后果，所以微软并不推荐使用这种方式，因此，本书也不讨论这个问题。

C#代码看起来更像是 Java 语言，常常比 C++略长一些。由于 C#的类型安全性，一旦为某些数据指定了类型，就不能转换为另一个不相关的类型。所以，在类型之间转换时，必须遵守严格的规则。执行相同的任务时，用 C#编写的代码通常比 C++长。但 C#代码更健壮，调试也比较简单，.NET 总是可以随时跟踪数据的类型。

相对于 C、C++等语言，C#具有以下优点：

- 简单。C#具有 C++所没有的一个优势就是学习简单。该语言首要的目标就是简单。在 C#中，没有 C++中流行的指针，而学习过 C 或 C++的读者都应该知道，指针是 C/C++学习中最难掌握的知识点。在默认情况下，C#编写的代码是受托管的，在那里不允许如直接存取内存等不安全的操作。
- 现代。C#可以直接使用.NET Framework 中强大的类库，很多用 C++可以实现或者很费力实现的功能，在 C#中不过是一部分基本的功能而已。C#中抛弃了指针，而且也不用关心内存管理，而是由垃圾收集器负责 C#程序中的内存管理，不用担心内存泄漏。因内存和应用程序都受到管理，所以很必要增强类型安全，以确保应用的稳定性。
- 面向对象。C#是一个完全面向对象（OO）的语言，在 C#中一切皆为对象，支持所有关键的面向对象的概念，如封装、继承和多态性。C#抛弃了 C++中多重继承的概念，避免了由于多重继承引起的麻烦。另外 C#在 2.0 中还添加了匿名函数、泛型等特性，在 3.0 中添加了对 Lambda 表达式等特性支持。
- 类型安全。C#中不能使用没有初始化的变量，取消了不安全的类型转换。不能把一个整型强制转换成一个引用类型（如对象），而当向下转换时，C#验证这种转换是正确的。在 C#中，被传递的引用参数是类型安全的。
- 版本可控。在过去的几年中，几乎所有的程序员都可能碰到众所周知的“DLL 地狱”。该问题起因于多个应用程序都安装了相同 DLL 名字的不同版本。如果新版本的 DLL 没有向老版本的 DLL 兼容，那么使用老版本 DLL 的程序可能就会抛出异常，中断运行。版本问题也便成了令人头痛的问题。而 C#可以很好地支持版本控制。C#可以为程序员保证版本控制成为可能。有了这种支持，一个开发人员就可以确保当他的类库升级时，仍保留着对已存在的客户应用程序的二进制兼容。
- 兼容。C#不仅仅可以调用使用.NET 语言编写的程序集，对于 C++、Visual Basic 等程序编写的 API 也可以在 C#中进行调用。C#允许使用通用语言规定（Common Language Specification，CLS）访问不同的 API。CLS 规定了一个标准，用于符合这种标准语言的内部之间的操作。
- 灵活。虽然前面说到 C#中不使用指针，类型是安全的，但是出于灵活性的考虑，仍然支持用户使用指针编写类型不安全的代码，但是用户必须声明类的方法是非安全类型的。在不安全的声明中允许用户使用指针、结构，静态地分配数组。

C#只是.NET 开发的一种语言，是专门为.NET Framework 设计的语言，所以能够很好地体现.NET 的特性。C#是在移植到其他操作系统上的.NET 版本中使用的主要语言，也是进行.NET 开发中的首推语言。要使语言如 VB.NET 尽可能类似于其以前的语言，且仍遵

循 CLR，就不能完全支持.NET 代码库的某些功能。但 C#能使用.NET Framework 代码库提供的每种功能。.NET 的最新版本还对 C#语言进行了几处改进，这是为了满足开发人员的要求，使之更强大。

若使用 C#进行开发，最重要的开发工具是微软的 VS。目前 VS 的最新版本是 2012（11.0 版），支持.NET Framework 2.0、3.0、3.5、4.0 和 4.5 的开发。为了能够完全使用到 SQL Server 2012 中的一些新特性，笔者推荐使用 VS 2012 进行开发。

9.1.3　CLR 集成概述

随着 SQL Server 编程技术的发展和成熟，代码编写人员陷入了 SQL Server 自身的一些限制之中，SQL Server 提供的存储过程和函数等十分有限，很多时候在很大程度上依赖外部的代码来执行一些繁重的移植。

T-SQL 作为数据查询语言，在返回数据集方面很好，但是除了这个之外则表现不佳。SQL Server 与 CLR 的集成解决了这方面的问题，使得原本需要一个完全独立的程序来实现的功能可以迁移到 SQL Server 内部进行数据操作。.NET 的操作代码和执行速度比 T-SQL 好得多。.NET 程序是二进制的，而不是作为存储过程来构建，所以不需要编译就可以运行。

SQL Server 与 CLR 的集成特别注重 CLR 的安全，所有的代码都在运行前检测类型和安全权限。例如，CLR 不能随便访问网络和磁盘文件等。通过.NET 框架实现的功能都可以通过 CLR 集成的方式，可以从存储过程、触发器或者用户函数进行访问，否则 CLR 在 SQL Server 是不会有用的。

为了防止 CLR 代码胡乱运行，任意访问其他资源，微软为 CLR 代码的调用创建了一个 3 层的安全模型：SAFE、EXTERNAL_ACCESS 和 UNSAFE。下面简要介绍一下这 3 层安全模型。

- SAFE：权限集合在本质上与传统的存储过程能够做的事情一样。在 SQL Server 之外不能对其进行任何修改。
- EXTERNAL_ACCESS：允许通过.NET 对注册表和文件系统进行访问。
- UNSAFE：正如其名，标记为 UNSAFE 的代码可以做任何事情，可以访问所有资源，所以实际上是不应该在调试或者实验环境之外使用 UNSAFE 模式。

一般来说，CLR 应该只是做数据处理，不应该访问外部资源，开发人员编写的 CLR 集成程序集应该永远不需要用到高于 EXTERNAL_ACCESS 级别的任何东西。如果需要在存储过程或者函数的环境中与文件系统或者注册表对话，则说明当前的处理逻辑有问题，需要重新考虑实现的逻辑。

CLR 集成一个很大的缺点是需要额外编程，CLR 中的逻辑被编译成程序集添加到 SQL Server 中，不易进行管理和更改。所以 CLR 集成适合于那些不容易、需要进行编程，必须在 T-SQL 中实现的环境。

大多数情况下简单的操作可以在 T-SQL 中以存储过程的方式完成，并且不需要扩展到外部进程。如果在操作过程中引用了 CLR 集成的存储过程则需要上下文交换和额外的事务开销，这两项中的任何一项开销都能首先抵消使用 CLR 获得的速度提升。所以使用 CLR 集成和 T-SQL 实现相同的逻辑功能，有可能 CLR 集成的效率更低。

CLR 最好用于替代扩展存储过程。例如，对于必须封闭在数据库中，但是使用 T-SQL 实现却非常麻烦，而使用程序实现却很简单，同时又不能轻松移动到业务逻辑末尾的事情。

CLR 集成还有另一个可能的缺点就是，如果将业务逻辑中的某个元素移动到数据库，那就可能会引起可测量性的问题。T-SQL 简洁、有效，而 CLR 集成昂贵但功能强大。由于 CLR 集成的程序集是以单独的文件存在于 SQL Server 之外的文件系统中，而不是像 T-SQL 一样存储在数据库系统中，所以对 CLR 程序集的管理也是一个麻烦的问题。

将 CLR 代码编译成 DLL 文件，然后注册到 SQL Server 中作为数据库对象，然后执行数据库操作，其过程包括以下步骤。

（1）开发人员将托管程序（比如 C#）编写为一组类定义。将 SQL Server 内本来用作存储过程、函数或触发器（下面统称为例程）的代码编写为类的 static（或 Microsoft Visual Basic .NET 中的 shared）方法。而对于用作用户定义的类型和聚合的代码编写为一个结构体。代码编写完成后将代码编译并创建一个 DLL 程序集。

（2）将此程序集上传到 SQL Server 数据库服务器磁盘上，然后使用 CREATE ASSEMBLY 数据定义语言将 DLL 程序集存储到系统目录。

（3）创建 SQL 对象，例如函数、存储过程、触发器、类型和聚合，并将其绑定到已经上载的程序集中的入口点（对例程来说是方法，对类型和聚合来说是类）。使用 CREATE PROCEDURE/FUNCTION/TRIGGER/TYPE/AGGREGATE 语句来完成这一步。

（4）在创建了这些例程之后，应用程序就可以像使用 T-SQL 例程一样使用它们。例如，可以从 T-SQL 查询中调用 CLR 函数，从客户端应用程序或从 T-SQL 批处理中调用 CLR 过程，就好像它们是 T-SQL 过程一样。

VS 2005 以后的版本支持在 SQL Server 中开发、部署和调试托管代码。VS 有一种新的项目类型（称为 SQL Server 项目），它允许开发人员在 SQL Server 中开发、部署和调试例程（函数、过程和触发器）、类型和聚合。

SQL Server 项目提供了代码模板，这使得开发人员能够轻松地开始为基于 CLR 的数据库例程、类型和聚合编写代码。该项目还允许添加对数据库中其他程序集的引用。在构建项目时，可以将其编译成一个程序集。使用 VS 部署程序集，可以将程序集的二进制文件上载到与该项目相关联的 SQL Server 数据库中。部署操作还自动创建在数据库的程序集中定义的例程、类型和聚合。另外，VS 还上载与该程序集相关联的源代码和.pdb 文件（调试符号），用于帮助开发人员调试 CLR 中的逻辑。

9.2　使用 CLR 集成的命名空间

Microsoft.SqlServer.Server 命名空间包含将 Microsoft.NET Framework 公共语言运行库（CLR）集成到 SQL Server 和 SQL Server 数据库引擎进程执行环境时所要用到的类、接口和枚举。

Microsoft.SqlServer.Server 命名空间下提供了多个与 CLR 集成相关的属性（Attribute）类，可以用任何支持的.NET Framework 的语言创建存储过程、触发器、用户定义的类型、用户定义的函数（标量值和表值）和用户定义的聚合函数。这里笔者推荐使用 C#语言创建 CLR 集成的相关代码。

在 CLR 集成编程中有一个非常重要的类就是 SqlContext。通过查询该类可确定当前执

行的代码是否在 SQL Server 数据库引擎进程中运行。当用户调用服务器上的托管存储过程或函数时，或者当用户操作激发托管代码触发器时，此查询还会提供调用方的上下文。

在 SqlContext 类中提供了一个 SqlPipe 对象，结果通过该对象从存储过程返回客户端；提供了一个 SqlTriggerContext 对象，该对象提供有关激发触发器操作的信息；还提供了一个 WindowsIdentity 对象，该对象可用于在客户端使用身份验证的集成安全性时确定调用客户端的标识。该命名空间下的具体类和说明如表 9.1 所示。

表 9.1　Microsoft.SqlServer.Server命名空间下提供的类

类	说　明
InvalidUdtException	异常类，在 SQL Server 或 ADO.NET System.Data.SqlClient 提供程序检测到无效的用户定义类型时引发
SqlContext	表示调用方上下文的抽象，该上下文提供对 SqlPipe、SqlTriggerContext 和 WindowsIdentity 对象的访问。无法继承此类
SqlDataRecord	表示单个数据行及其元数据。无法继承此类
SqlFacetAttribute	使用可用在 T-SQL 中的其他信息，对用户定义类型的返回结果进行批注
SqlFunctionAttribute	用于将用户定义聚合的方法定义标记为 SQL Server 中的函数
SqlMetaData	从 SqlDataRecord 对象的参数和列指定、检索元数据信息。无法继承此类
SqlMethodAttribute	指示用户定义类型的方法或属性的确定性和数据访问性质
SqlPipe	允许托管存储过程在 SQL Server 数据库上进行进程内运行，以便将结果返回调用方。无法继承此类
SqlProcedureAttribute	用于将程序集中的方法定义标记为存储过程
SqlTriggerAttribute	用于将程序集中的方法定义标记为 SQL Server 中的触发器
SqlTriggerContext	提供所激发的触发器的上下文信息
SqlUserDefinedAggregateAttribute	指示类型应注册为用户定义的聚合。无法继承此类
SqlUserDefinedTypeAttribute	用于将程序集中的类型定义标记为 SQL Server 中的用户定义类型。无法继承此类

所有用于和 SQL Server 集成的 CLR 程序集都必须添加对该命名空间的引用，要创建不同的数据库对象则使用该命名空间下对应的类：

- 存储过程需要添加 SqlProcedureAttribute。
- 函数需要添加 SqlFunctionAttribute。
- 用户定义类型需要添加 SqlUserDefinedTypeAttribute。
- 触发器需要添加 SqlTriggerAttribute。
- 聚合需要添加 SqlUserDefinedAggregateAttribute。

9.3　SQL Server 中的程序集

使用.NET 平台语言编写的程序（DLL 文件）可以以程序集的形式添加到 SQL Server 中。在添加了程序集后才能以程序集为基础建立 CLR 存储过程和 CLR 函数等。本节将主要讲解程序集的创建和管理。

9.3.1　程序集简介

程序集是在 SQL Server 实例中使用的 DLL 文件，用来部署用 Microsoft .NET Framework 公共语言运行时（CLR）中所驻留的托管代码语言之一（而非 Transact-SQL）编写的函数、存储过程、触发器、用户定义聚合和用户定义类型。

程序集作为 SQL Server 中的数据库对象，主要负责引用.NET Framework 公共语言运行时中创建的托管应用程序模块（.dll 文件）。程序集包含类元数据和托管代码。创建程序集是 CLR 集成在 SQL Server 上操作的第一步，将程序集上载到 SQL Server 实例可以创建以下数据库对象：

- CLR 函数。
- CLR 存储过程。
- CLR 触发器。
- 用户定义聚合函数。
- 用户定义类型。

简单地说，在 SQL Server 范围内，程序集是一个引用物理程序集.dll 文件的对象。受管代码是.dll 文件，该文件使用.NET Framework CLR 和可访问其他受管代码来创建。更确切地说，是在 SQL Server 内部的其他受管代码。

在程序集中的每段受管代码都包括两个重要的片段信息。一个是描述程序集的元数据，例如程序集方法和属性，程序集版本号。第二个片段信息是实际的受管代码，组成程序集的方法和属性。通常使用 C#编写托管代码，这些代码共享类库，同时被编译为中间语言（Intermediate Language，IL）。

程序集中的受管代码定义了要实现的 SQL Server 对象的功能，例如存储过程、UDT、CLR 函数和 CLR 触发器。程序集自身控制受管代码访问内部和外部资源的权限级别。当在 SQL Server 中利用 CREATE ASSEMBLY 语句创建程序集时，.dll 文件会物理地加载到 SQL Server 中，这样 SQL Server 引擎就能够引用和使用程序集。

9.3.2　使用 T-SQL 添加程序集

当已经准备好了 dll 文件，开始在 SQL Server 2012 中使用程序集之前，需要告知 SQL Server 已经准备好在 SQL Server 中与 CLR 交互。出于安全的考虑，SQL Server 2012 默认情况下禁用 CLR 集成功能，必须启用 CLR 集成才能在 SQL Server 中访问.NET 对象。为了启用 CLR 集成，需要使用 T-SQL 开启该配置。具体 SQL 脚本如代码 9.1 所示。

代码 9.1　启用 CLR 集成

```
--在 SQL Server 中执行这段代码可以开启 CLR
EXEC sp_configure 'show advanced options', '1';
GO
RECONFIGURE;
GO
EXEC sp_configure 'clr enabled', '1' --开启 CLR
GO
```

```
RECONFIGURE
```

启用 CLR 集成后，接下来需要将 dll 文件作为程序集添加到 SQL Server 中。在 SQL Server 中要添加程序集需要使用 CREATE ASSEMBLY 命令，其语法如代码 9.2 所示。

代码 9.2　CREATE ASSEMBLY 语法

```
CREATE ASSEMBLY assembly_name
[ AUTHORIZATION owner name ]
FROM { <client assembly specifier> | <assembly bits> [ ,...n ] }
[ WITH PERMISSION SET = { SAFE | EXTERNAL ACCESS | UNSAFE } ]
[ ; ]
<client assembly specifier> :: =
'[\\computer name\]share name\[path\]manifest file name'
| '[local path\]manifest file name'
<assembly bits> :: =
{ varbinary_literal | varbinary_expression }
```

其中比较重要的几个参数介绍如下。

- assembly_name：程序集的名称。此名称必须在数据库中唯一，并且是有效的标识符。
- <client_assembly_specifier>：指定正在上传的程序集所在的本地路径或网络位置，以及与程序集对应的清单文件名。<client_assembly_specifier> 可表示为固定字符串或其值等于固定字符串的、带有变量的表达式。CREATE ASSEMBLY 不支持加载多模块程序集。SQL Server 还将在同一位置查找此程序集的所有相关程序集，并使用与根级别程序集相同的所有者将其上传。如果没有找到这些相关程序集且它们尚未加载到当前数据库中，则 CREATE ASSEMBLY 失败。如果相关程序集已加载到当前数据库中，则这些程序集的所有者必须与新创建的程序集的所有者相同。如果模拟的是登录用户，则无法指定<client_assembly_specifier>。
- <assembly_bits>：组成程序集和依赖程序集的二进制值的列表。列表中的第一个值将视为根级程序集。与相关程序集对应的值可以按任意顺序提供。所有与根程序集的依赖项不相对应的值都将忽略。
- PERMISSION_SET { SAFE | EXTERNAL_ACCESS | UNSAFE }：指定 SQL Server 访问程序集时向程序集授予的一组代码访问权限。如果未指定，则将 SAFE 用做默认值。

大多数情况下都是使用 SAFE，其是最具限制性的权限集。由具有 SAFE 权限的程序集所执行的代码将无法访问外部系统资源，例如文件、网络、环境变量或注册表。而使用 EXTERNAL_ACCESS 则允许程序集访问某些外部系统资源，例如文件、网络、环境变量以及注册表。UNSAFE 下则可使程序集不受限制地访问资源，无论是 SQL Server 实例内部还是外部的资源都可以访问。从 UNSAFE 程序集内运行的代码可调用未托管代码。

注意：即使是在 UNSAFE 模式下程序集对资源的访问仍然受到 Windows 账户权限的控制，在 Windows 下 SQL Server 账户无权访问的资源程序集也无法访问。例如在 Windows 下禁止 SQL Server 账户访问 C:\Data 文件夹，则运行在 UNSAFE 模式下的程序集就无法访问该文件夹。

假设现在有程序集 SqlServerProject1.dll 位于 D:\Lib 目录下，若要将该程序集添加到数

据库 TestDB1 中，并且使用安全权限设置，则对应的 SQL 脚本如代码 9.3 所示。

代码 9.3　创建程序集

```
USE TestDB1;
GO
CREATE ASSEMBLY TestSQLAssembly--程序集名
AUTHORIZATION dbo
FROM 'D:\Lib\SqlServerProject1.dll' --程序集地址
WITH PERMISSION_SET=SAFE
```

创建程序集后可以通过查询 sys.assemblies 获得当前数据库中的所有程序集的信息。

9.3.3　使用 SSMS 添加程序集

若要在 SSMS 下通过可视化操作创建程序集则相对简单得多，具体操作如下所述。

（1）使用 SSMS 连接到数据库后在对象资源管理器中依次展开“数据库”、“TestDB1”、“可编程性”、“程序集”等节点。右击“程序集”节点，在弹出的快捷菜单中选择“新建程序集”命令，系统将弹出“新建程序集”对话框，如图 9.1 所示。

（2）由于这里需要创建的是安全类型的程序集，所以在“权限集”下拉列表框中选择安全选项。

（3）单击“浏览”按钮，系统将弹出选择程序集路径的对话框。通过该对话框找到需要的程序集，系统将根据程序集的文件名自动命名程序集名称，该名称就是引用的 dll 的文件名而且不可修改。

（4）单击“确定”按钮，系统将完成程序集的创建，同时在对象资源管理器中也可以看到当前创建的程序集，如图 9.2 所示。

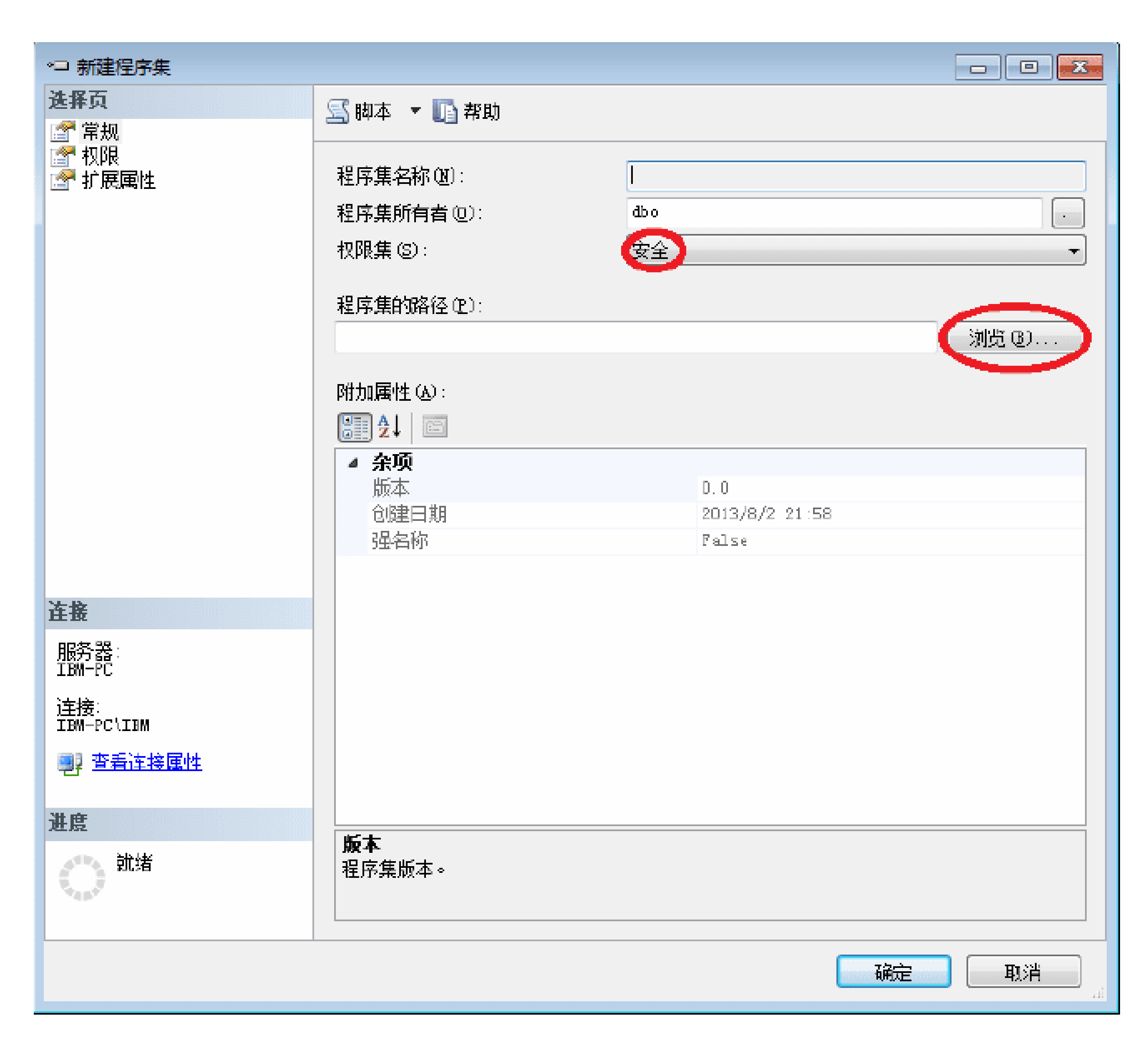


图 9.1　“新建程序集”对话框

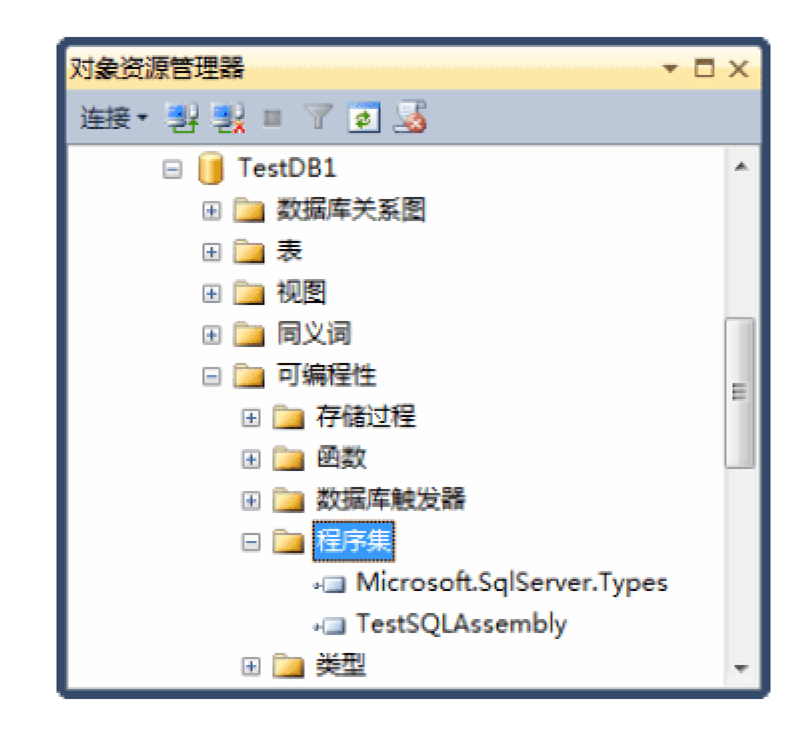


图 9.2　查看已有的程序集

注意：相同的 dll 文件只能在同一个数据库中创建一个程序集，即使将同一个 dll 改名或移动到其他位置系统也只能为其创建一个程序集。

9.3.4　修改程序集

程序集创建后若更新了 CLR 程序，则需要修改程序集。修改程序集使用 ALTER ASSEMBLY 命令，其语法如代码 9.4 所示。

代码 9.4　ALTER ASSEMBLY 语法

```
ALTER ASSEMBLY assembly name
    [ FROM <client_assembly_specifier> | <assembly_bits> ]
    [ WITH <assembly option> [ ,...n ] ]
    [ DROP FILE { file name [ ,...n ] | ALL } ]
    [ ADD FILE FROM
    {
              client file specifier [ AS file name ]
      | file bits AS file name
    } [,...n ]
    ] [ ; ]
<client assembly specifier> :: =
       '\\computer name\share-name\[path\]manifest file name'
  | '[local path\]manifest file name'

<assembly bits> :: =
    { varbinary literal | varbinary expression }

<assembly_option> :: =
    PERMISSION SET = { SAFE | EXTERNAL ACCESS | UNSAFE }
  | VISIBILITY = { ON | OFF } ]
  | UNCHECKED DATA
```

其参数与创建程序集时使用的参数基本相同，这里只介绍一下只是 ALTER ASSEMBLY 命令才使用的参数。

- VISIBILITY={ON|OFF}：指示在创建 CLR 函数、存储过程、触发器、用户定义的类型以及针对它的用户定义的聚合函数时，该程序集是否可见。如果设置为 OFF，则程序集只能由其他程序集调用。
- UNCHECKED DATA：默认情况下，如果 ALTER ASSEMBLY 必须验证各个表行的一致性，则它将失败。该选项使得用户可以通过使用 DBCC CHECKTABLE 将检查推迟到以后的某个时间进行。
- [DROP FILE{file_name[,...n]|ALL}]：从数据库中删除与程序集关联的文件名，或与该程序集关联的所有文件。如果与下面的 ADDFILE 一起使用，则 DROPFILE 首先执行。这样可以用相同的文件名替换文件。
- [ADD FILE FROM{client_file_specifier[AS file_name]|file_bits AS file_name}：将与程序集关联的文件（如源代码、调试文件或其他相关信息）上载到服务器中并使其在 sys.assembly_files 目录视图中可见。client_file_specifier 指定上载文件的位置。可以改用 file_bits 来指定构成该文件的二进制值列表。file_name 指定将文件存储

到 SQL Server 实例中时所采用的名称。如果指定了 file_bits，则必须指定 file_name；如果指定了 client_file_specifier，则该参数是可选的。如果未指定 file_name，则将 client_file_specifier 的 file_name 部分用做 file_name。

注意：如果执行了没有 UNCHECKED 数据子句的 ALTER ASSEMBLY，则系统会执行检查以验证新数据集版本是否影响表中的现有数据。根据需要检查的数据量，可能会影响性能。

以前面创建的 TestSQLAssembly 程序集为例，现在若需要将该程序集的 pdb 调试文件添加到该程序集中，则对应的 SQL 脚本如代码 9.5 所示。

代码 9.5　添加文件

```
USE [TestDB1]
GO
ALTER ASSEMBLY TestSQLAssembly --程序集的名字
ADD FILE FROM 'D:\Lib\SqlServerProject1.pdb' --添加文件
```

若 CLR 文件版本已经更新，需要更新 TestSQLAssembly 程序集，同时又要添加其调试文件，则对应的 SQL 脚本如代码 9.6 所示。

代码 9.6　更新程序集

```
USE [TestDB1]
GO
ALTER ASSEMBLY TestSQLAssembly --修改指定程序集
DROP FILE ALL --删除所有文件
GO
ALTER ASSEMBLY TestSQLAssembly
FROM 'D:\Lib\SqlServerProject1.dll'
ADD FILE FROM 'D:\Lib\SqlServerProject1.pdb' --添加文件
```

注意：对于已经添加了文件的程序集，若要更新 CLR 文件则必须要删除所有添加的文件，然后更新 CLR 程序和重新添加文件。

在 SSMS 中对程序集的修改十分有限，只能修改程序集的所有者和权限集。若要修改所有者或权限集，只需双击程序集名，在弹出的属性窗口中直接修改即可。

9.3.5　删除程序集

若程序集不再使用，需要将其删除，则使用 DROP ASSEMBLY 命令，其语法如代码 9.7 所示。

代码 9.7　DROP ASSEMBLY 语法

```
DROP ASSEMBLY assembly name [ ,...n ]
[ WITH NO DEPENDENTS ]
```

其中，assembly_name 就是要删除的程序集名，如果指定了 WITH NO DEPENDENTS，则只删除 assembly_name，而不删除该程序集引用的相关程序集。如果不指定它，则 DROP

ASSEMBLY 将删除 assembly_name 和所有相关程序集。

删除程序集时，将从数据库中删除程序集和它的所有关联文件，例如，源代码和调试文件。如果程序集被存在于该数据库中的另一个程序集引用，或者它被当前数据库中的 CLR 函数、过程、触发器、用户定义类型或聚合使用，则 DROP ASSEMBLY 返回错误。

DROP ASSEMBLY 不会干扰引用当前正在运行的程序集的任何代码。但是执行 DROP ASSEMBLY 之后，任何调用程序集代码的尝试将失败。例如要将前面创建的程序集 TestSQLAssembly 删除，则只需要运行脚本：

```
DROP ASSEMBLY TestSQLAssembly
```

在 SSMS 中删除程序集的操作与删除其他数据库对象的操作相同，在对象资源管理器中选中需要删除的程序，然后使用 Delete 快捷键，系统将弹出删除对象对话框，单击“确定”按钮即可。

9.4　创建 CLR 函数

CLR 函数就是将在 SQL Server 中创建的可在 Microsoft .NET Framework 公共语言运行时创建的程序集转换为 SQL Server 中的函数。使用 CLR 函数可以轻松完成在 T-SQL 中需要编写很复杂的 T-SQL 语句才能完成的工作。本节将从标量值函数和表值函数两个方面来讲解 CLR 函数的创建和使用。

9.4.1　使用 C#编写 CLR 标量值函数

前面已经讲到，若要创建被 SQL Server 引用的 CLR 程序则需要引用命名空间 Microsoft.SqlServer.Server。创建 CLR 函数需要使用到该命名空间下的 SqlFunctionAttribute 类。VS 2010 和 VS 2012 都提供了创建 SQL Server 程序集的项目模板，有助于程序员轻松地实现 SQL Server 程序集的创建。以 VS 2012 为例，在其中创建一个标量值函数，该实现输入一个字符串，返回过滤掉字符串中的 HTML 标签的字符串。创建函数的具体操作如下所述。

（1）打开 VS 2012，新建数据库项目中的 SQL Server Project 项目，项目名为 SqlServerProject1。

（2）添加用户定义函数 HtmlFilter()，系统将创建 C#代码，如代码 9.8 所示。

代码 9.8　系统创建的用户定义函数

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
//创建用户定义函数所在的类，该类的类名是 UserDefinedFunctions
public partial class UserDefinedFunctions
{
//通过 Attribute 来标明静态方法 HtmlFilter 可以作为 SQL 中的用户定义函数
```

```
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlString HtmlFilter()//函数名
    {
        //在此输入你的代码
        return new SqlString("Hello");   //返回结果
    }
};
```

其中系统已经添加了对 Microsoft.SqlServer.Server 命名空间的引用。同时也创建了函数 HtmlFilter()，该函数添加了属性 SqlFunction，用于标识该函数可以在创建 SQL Server 函数时被引用。

（3）修改 HtmlFilter()函数，实现过滤 HTML 标签的功能。修改后的代码如代码 9.9 所示。

代码 9.9　修改后的 HtmlFilter()函数

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text.RegularExpressions;
//创建用户定义函数所在的类，该类的类名是 UserDefinedFunctions
public partial class UserDefinedFunctions
{
//通过 Attribute 来标明静态方法 HtmlFilter 可以作为 SQL 中的用户定义函数
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlString HtmlFilter(SqlString html) //传送字符串参数 html
    {
        if (html.IsNull) //如果传入字符串为空，则返回空
        {
            return null;
        }
//返回经过处理的字符串
        return new SqlString(Regex.Replace(html.Value, "<[^>]+>", ""));
    }
}
```

（4）使用快捷键 Ctrl+Shift+B 编译整个项目，编译成功后就可以在项目的 bin 文件夹下的 Debug 文件夹中，找到编译出的 CLR 程序 SqlServerProject1.dll。

这里需要注意的是，C#代码中使用的是 SqlString 数据类型作为函数的输入和输出，若改为使用 string 也不会有影响，但笔者仍然推荐使用 SqlString 类型，这将具有更好的兼容性。同样地，若要使用布尔值、整数、日期时间等都有对应的 SqlBoolean、SqlInt32、SqlDateTime 等。详细的对照关系如表 9.2 所示。

表 9.2　SQL Server数据类型与CLR数据类型、.NET数据类型对应

SQL Server 数据类型	CLR 数据类型(SQL Server)	CLR 数据类型(.NET Framework)
bigint	SqlInt64	Int64, Nullable<Int64>
binary	SqlBytes, SqlBinary	Byte[]
bit	SqlBoolean	Boolean, Nullable<Boolean>
char	None	None

续表

SQL Server 数据类型	CLR 数据类型(SQL Server)	CLR 数据类型(.NET Framework)
Cursor	None	None
date	SqlDateTime	DateTime, Nullable<DateTime>
datetime	SqlDateTime	DateTime, Nullable<DateTime>
datetime2	SqlDateTime	DateTime, Nullable<DateTime>
datetimeoffset	None	DateTimeOffset, Nullable<DateTimeOffset>
decimal	SqlDecimal	Decimal, Nullable<Decimal>
float	SqlDouble	Double, Nullable<Double>
image	None	None
int	SqlInt32	Int32, Nullable<Int32>
money	SqlMoney	Decimal, Nullable<Decimal>
nchar	SqlChars, SqlString	String, Char[]
ntext	None	None
numeric	SqlDecimal	Decimal, Nullable<Decimal>
nvarchar	SqlChars, SqlString	String, Char[]
nvarchar(1), nchar(1)	SqlChars, SqlString	Char, String, Char[], Nullable<char>
real	SqlSingle	Single, Nullable<Single>
rowversion	None	Byte[]
smallint	SqlInt16	Int16, Nullable<Int16>
smallmoney	SqlMoney	Decimal, Nullable<Decimal>
sql_variant	None	Object
table	None	None
text	None	None
time	TimeSpan	TimeSpan, Nullable<TimeSpan>
timestamp	None	None
tinyint	SqlByte	Byte, Nullable<Byte>
uniqueidentifier	SqlGuid	Guid, Nullable<Guid>
User-defined type(UDT)	None	相同的类
Varbinary	SqlBytes, SqlBinary	Byte[]
varbinary(1), binary(1)	SqlBytes, SqlBinary	byte, Byte[], Nullable<byte>
Varchar	None	None
Xml	SqlXml	None

至此实现字符串中 HTML 标签过滤的 CLR 函数已经完成。接下来就是将生成的程序集添加到 SQL Server 中。

9.4.2　在 SQL Server 中使用 CLR 标量值函数

使用 CREATE ASSEMBLY 命令添加了 HtmlFilter()函数所在的程序集后，就可以使用该程序集创建 SQL Server 标量值函数。在 SQL Server 中使用 CREATE FUNCTION 语句创

建引用注册程序集的函数。其语法格式如代码 9.10 所示。

代码 9.10　创建 CLR 标量值函数的语法

```
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
)
RETURNS { return_data_type }
    [ WITH <clr_function_option> [ ,...n ] ]
    [ AS ] EXTERNAL NAME assembly_name.class_name.method_name
<clr_function_option>::=
}
    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
}
```

其语法格式与前面介绍的创建 T-SQL 标量值函数的格式相同，这里只介绍一下不同的两个参数。

- EXTERNAL NAME <method_specifier> assembly_name.class_name.method_name：指定将程序集与函数绑定的方法。assembly_name 必须与 SQL Server 中当前数据库内具有可见性的现有程序集匹配。class_name 必须是有效的 SQL Server 标识符，并且必须作为类存在于程序集中。如果类具有以命名空间限定的名称，该名称使用句点来分隔命名空间的各部分，则必须使用方括号或引号分隔类名称。method_name 必须是有效的 SQL Server 标识符，并且必须作为静态方法存在于指定类中。
- RETURNS NULL ON NULL INPUT|CALLED ON NULL INPUT：指定标量值函数的 OnNULLCall 属性。如果未指定，则默认为 CALLED ON NULL INPUT。这意味着即使传递的参数为 NULL，也将执行函数体。

如果在 CLR 函数中指定了 RETURNS NULL ON NULL INPUT，它指示当 SQL Server 接收到的任何一个参数为 NULL 时，它可以返回 NULL，而无需实际调用函数体。如果<method_specifier>中指定的 CLR 函数的方法已具有指示 RETURNS NULL ON NULL INPUT 的自定义属性，但 CREATE FUNCTION 语句指示 CALLED ON NULL INPUT，则优先采用 CREAT EFUNCTION 语句指示的属性。不能为 CLR 表值函数指定 OnNULLCall 属性。

将 CLR 程序 SqlServerProject1.dll 添加到 SQL Server 中作为程序集 TestSQLAssembly 后，创建 CLR 函数调用其中的 HtmlFilter 函数的脚本如代码 9.11 所示。

代码 9.11　创建 CLR 标量值函数

```
CREATE FUNCTION [dbo].[HtmlFilter] -SQL 中的函数名
(
@html [nvarchar](1000) --CLR 标量值函数的参数
)
RETURNS [nvarchar](200)  --CLR 标量值函数的返回值
AS --以下定义函数对应的程序集中的位置
EXTERNAL NAME [TestSQLAssembly].[UserDefinedFunctions].[HtmlFilter]
```

函数创建后便可在 T-SQL 语句中应用该函数，与普通的 T-SQL 创建的函数在使用上没有任何不同。使用 CLR 标量值函数的脚本示例如代码 9.12 所示。

代码 9.12　使用 CLR 标量值函数

```
select dbo.HtmlFilter('<a href="about.aspx">About Us</a>') --调用 CLR 标量值函数
--返回无 HTML 标签的结果:
About Us
```

9.4.3　使用 C#编写 CLR 表值函数

CLR 表值函数返回的是一个表，在.NET 中创建对应的函数，返回的结果是一个 IEnumerable 接口，用于表示一个集合。这个集合中是对象的实例并不是为 SQL Server 所识别的表，所以需要在函数的属性中指定 FillRowMethodName，这个参数的值是用于将.NET 中的对象转换为表列的函数名。

以字符串分割函数为例，C#提供了 Split 函数用于指定分隔符将字符串分割为字符串数组，但是 SQL Server 中却没有提供类似的函数。该函数可以使用 T-SQL 实现，但是相对比较复杂而且处理效率不高，这里就用 CLR 表值函数的形式来实现字符串分割。在 SqlServerProject1 项目中添加用户定义函数，使用 C#创建用于表值函数的代码，如代码 9.13 所示。

代码 9.13　StringSplit()函数

```
using System;
using System.Data;
using System.Collections;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
//创建用户定义函数所在的类，该类的类名是 UserDefinedFunctions
public partial class UserDefinedFunctions
{
//使用 Attribute 指定 StringSplit 函数是 CLR 表值函数
//FillSplitTable 是将.NET 中的对象转换为表列的函数名
    [Microsoft.SqlServer.Server.SqlFunction(FillRowMethodName=
    "FillSplitTable")]
    public static IEnumerable  StringSplit(SqlString str,SqlString split)
                                            //定义函数
    {
        if (str.IsNull || split.IsNull) //如果任何一个参数为空，则返回空表
        {
            return "".ToCharArray();
        }
        return str.Value.Split(split.Value.ToCharArray());
                                            //两个参数都不为空，返回字符串分割结果
    }
    //这里定义了将.NET 中的对象转换为表列的静态方法
    public static void FillSplitTable(object obj, ref SqlString F1)
                                            //F1 就是返回表的列名
```

```
    {
        string s = obj.ToString();          //传入的对象
        F1 =new SqlString(s);               //将字符串 s 作为 F1 列输出
    }
}
```

从代码中可以看出返回了一个 IEnumerable 接口，同时指定了使用函数 FillSplitTable()来进行解析。用于将.NET 对象转换为 SQL 列的 FillSplitTable()方法必须是静态方法，第一个参数必须属于 System.Object 类型。接下来的参数的个数就是列的个数，同时接下来的参数都必须申明为 ref，由于这里只输出一列，所以 FillSplitTable()函数总共只有 2 个参数。除了数目上相同外，SQL Server 中返回的列的数据类型和顺序必须与该函数中 ref 参数的数据类型和顺序相同。

添加了该函数后重新编译代码生成 SqlServerProject1.dll 文件，该文件用于更新当前 SQL Server 程序集中的 dll。

9.4.4　在 SQL Server 中使用 CLR 表值函数

重新生成了 dll 文件后，若要在 SQL Server 中使用 StringSplit()函数，必须先更新程序集。更新程序集使用 ALTER ASSEMBLY 命令，这里更新 TestSQLAssembly 程序集的 SQL 脚本如代码 9.14 所示。

代码 9.14　更新程序集

```
ALTER ASSEMBLY [TestSQLAssembly] --更新程序集
FROM 'D:\Lib\SqlServerProject1.dll'
WITH PERMISSION_SET = SAFE
```

更新程序集后便可创建使用 StringSplit()函数的 CLR 表值函数。创建 CLR 表值函数仍然使用 CREATE FUNCTION 命令，但是与创建 CLR 标量值函数语法有所不同。其语法如代码 9.15 所示。

代码 9.15　创建 CLR 表值函数的语法

```
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
)
RETURNS TABLE <clr_table_type_definition>
    [ WITH <clr_function_option> [ ,...n ] ]
    [ ORDER( <order_clause> )]
    [ AS ] EXTERNAL NAME assembly_name.class_name.method_name
[ ; ]
<clr_table_type_definition>::=
( { column_name data_type } [ ,...n ] )
<clr_function_option>::=
{
    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
}
<order_clause> ::=
{
```

```
   <column name in clr table type definition>
   [ ASC | DESC ]
} [ ,...n]
```

其中大部分的参数在创建用户定义函数和创建 CLR 标量值函数中已经进行了讲解，对于创建 CLR 表值函数，需要注意的是：

< clr_table_type_definition > ({ column_name data_type } [,...n])定义 CLR 函数的表数据类型。表声明仅包含列名称和数据类型。表始终放在主文件组中。

ORDER (<order_clause>)指定从表值函数中返回结果的顺序。

在 CLR 表值函数中使用 ORDER 子句时要遵循以下准则：

- 必须确保始终按指定的顺序对结果进行排序。如果没有按指定的顺序对结果进行排序，则在执行查询时 SQL Server 将生成一条错误消息。
- 如果指定了 ORDER 子句，则必须根据列（显式或隐式）的排序规则对表值函数的输出进行排序。例如，如果列排序规则为中文（在表值函数的 DDL 中指定或从数据库排序规则中获取），则必须根据中文排序规则对返回的结果进行排序。
- 如果指定了 ORDER 子句，则在返回结果时始终由 SQL Server 验证 ORDER 子句，而不管查询处理器是否会使用该子句执行进一步的优化。只有知道 ORDER 子句对查询处理器有用时才使用它。

在以下情况下，SQL Server 查询处理器将自动使用 ORDER 子句：

- 其中 ORDER 子句与索引兼容的“插入”查询。
- 与 ORDER 子句兼容的 ORDER BY 子句。
- 其中 GROUP BY 与 ORDER 子句兼容的聚合。
- 其中不同列与 ORDER 子句兼容的 DISTINCT 聚合。

除非在查询中还指定了 ORDER BY，否则 ORDER 子句不保证在执行 SELECT 查询时得到有序结果。

将程序集中的 StringSplit 函数创建为 SQL Server 表值函数 StringSplit()，那么其创建脚本如代码 9.16 所示。

代码 9.16　创建 StringSplit 表值函数

```
CREATE FUNCTION StringSplit --创建 CLR 表值函数
(
    @str nvarchar(4000),
    @split nvarchar(10)
)
RETURNS TABLE([F1] nvarchar(4000)) 返回一个表，表中只有一个列 F1
AS --以下定义函数对应的 CLR 中的位置
EXTERNAL NAME [TestSQLAssembly].[UserDefinedFunctions].StringSplit
```

创建该表值函数后调用该函数和返回结果如代码 9.17 所示。

代码 9.17　使用 CLR 表值函数

```
SELECT *
FROM dbo.StringSplit('a,bc,def,g',',')  --使用逗号分隔字符串
--返回结果为：
a
bc
```

```
def
g
```

9.5　创建 CLR 存储过程

存储过程在 SQL Server 中得到了广泛的应用，大多数数据库应用程序都是直接与存储过程交互完成工作。创建 CLR 存储过程的方法与创建 CLR 函数的方法相同，本节将主要讲解如何创建和使用 CLR 存储过程。

9.5.1　使用 C#编写 CLR 存储过程所需的函数

在 C#中编写可用于 CLR 存储过程引用的函数必须使用 SqlProcedure 属性标识。存储过程不像函数那样需要返回值，所以一般在 C#中建立 void 函数即可。存储过程一般用于查询并生成一个查询的表，在 C#中需要使用 SqlPipe 对象将表格结果与信息传回给客户端。

一般来说，通过 SqlContext 类的 Pipe 属性取得 SqlPipe 对象，然后调用 SqlPipe 对象的 Send()方法将表格结果或信息传送给客户端。SqlPipe 对象的 Send 方法提供了 3 个重载：

- SqlPipe.Send(string message)方法能够将字符串信息直接传送至客户端。文字最长不可超过 8000 个字符，超出 8000 个字符的部分将会被截掉。
- SqlPipe.Send(SqlDataRecord record)方法能够将单一数据列结果集（也就是一个 SqlDataRecord 对象）直接传送至客户端。
- SqlPipe.Send(SqlDataReader reader)方法能够将一个多数据列结果集（也就是一个 SqlDataReader 对象）直接传送至客户端。

注意：如果 Send()方法中所使用的 SqlDataReader 对象拥有隐藏字段，这些字段将不会填入传送给客户端的结果集中。

除了使用 Send()方法之外，CLR 存储过程还可以使用 SqlPipe 对象的 ExecuteAndSend()方法将查询结果传送给客户端。ExecuteAndSend()方法的最大好处就是，它提供了一种最高效率的方式将查询结果传送给客户端。之所以如此，是因为数据是通过网络缓冲区来传送，而不需被复制到受管理的内存中。以下是 ExecuteAndSend()方法的语法：

```
SqlPipi.ExecuteAndSend (command As SqlCommand)
```

从语法中可以看出，ExecuteAndSend()方法其实是以一个 SqlCommand 对象作为其参数，它会执行 SqlCommand 对象并将结果传送至客户端。除了所有实际的执行结果之外，其他的信息与错误也会直接传送给客户端。

例如，要创建一个对用户打招呼的 CLR 存储过程，传入参数@name，输出结果为“Hello”加上传入的姓名。同样使用前面创建的项目 SqlServerProject1，在该项目中添加存储过程类，最终 C#代码如代码 9.18 所示。

代码 9.18　用于 CLR 存储过程的 C#代码

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
//定义 CLR 存储过程所在的类，该类的类名是 StoredProcedures
public partial class StoredProcedures
{
//使用该 Attribute 标明函数 SayHello 是作为 CLR 存储过程被调用的
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void SayHello(SqlString name)//定义静态函数作为 CLR 存储过程
对应函数
    {
    //定义返回的列名和数据类型
        SqlMetaData data=new SqlMetaData("Say",SqlDbType.NVarChar,100);
        SqlDataRecord record = new SqlDataRecord(new SqlMetaData[] { data });
        if (!name.IsNull)
        {
            record.SetString(0, "Hello," + name.Value);
                                            //设置第一行返回的行内容
        }
        SqlContext.Pipe.Send(record);       //将返回的结果集发送给客户端
    }
};
```

编译整个项目为 SqlServerProject1.dll 文件，然后更新 SQL Server 中的程序集。

9.5.2　在 SQL Server 中使用 CLR 存储过程

更新完程序集后，便可以创建存储过程来调用其中的 SayHello()方法。创建 CLR 存储过程的语法如代码 9.19 所示。

代码 9.19　创建 CLR 存储过程的语法

```
CREATE { PROC | PROCEDURE } [schema name.] procedure name [ ; number ]
    [ { @parameter [ type_schema_name. ] data_type }
        [ VARYING ] [ = default ] [ OUT | OUTPUT ] [READONLY]
    ] [ ,...n ]
[ WITH <procedure option> [ ,...n ] ]
[ FOR REPLICATION ]
AS EXTERNAL NAME assembly name.class name.method name
[;]
<procedure_option> ::=
    [ ENCRYPTION ]
    [ RECOMPILE ]
    [ EXECUTE AS Clause ]
```

其中 EXTERNAL NAME assembly_name.class_name.method_name 指定.NET Framework 程序集的方法，以便 CLR 存储过程引用。class_name 必须为有效的 SQL Server 标识符，并且该类必须存在于程序集中。指定的方法必须为该类的静态方法。

前面编写的用于问候的函数 SayHello()已经被添加到 TestSQLAssembly 程序集中，该

函数只接受一个字符串参数，所以对应的创建 CLR 存储过程的脚本如代码 9.20 所示。

代码 9.20　创建 CLR 存储过程

```
CREATE  PROCEDURE dbo.SayHello --创建 CLR 存储过程
(
@name nvarchar(10)
)
--以下指定 CLR 存储过程所在程序集中的位置
AS EXTERNAL NAME [TestSQLAssembly].[StoredProcedures].SayHello
```

创建该存储过程后，使用 EXEC 便可调用该存储过程，系统调用和返回结果如代码 9.21 所示。

代码 9.21　调用 CLR 存储过程

```
EXEC dbo.SayHello @name='ZengYi'
系统返回结果：
Say
---------------
Hello,ZengYi
```

9.5.3　创建有 OUTPUT 参数的 CLR 存储过程

存储过程中可以使用带有 OUTPUT 的参数，带有 OUTPUT 的参数值在存储过程内部被修改后也会将修改应用到存储过程外部，相当于 C++中的指针或 C#中的 ref 参数。

对于 OUTPUT 类型的参数，在 C#中对应的就是 ref 类型的参数，例如要创建一个加法存储过程，输入 a、b、c 为 OUTPUT 类型，执行后 c 参数就是 a 和 b 的和。首先在 C#中实现的函数如代码 9.22 所示。

代码 9.22　C#中实现加法存储过程的函数

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
//定义 CLR 存储过程所在的类，该类的类名是 StoredProcedures
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
                                           //标明 Add 静态函数是 CLR 存储过程用
    public static void Add(SqlInt32 a,SqlInt32 b,ref SqlInt32 c)//定义函数
    {
        if (a.IsNull || b.IsNull)          //如果 a 或 b 为空，则返回空
            c = SqlInt32.Null;
        else                               //将 a+b 的值赋给 c
            c = new SqlInt32(a.Value + b.Value);
    }
};
```

重新编译并更新程序集，在 SQL Server 中创建用于加法的存储过程 AddNumber。创

建和使用该存储过程如代码 9.23 所示。

代码 9.23　创建和使用带 OUTPUT 参数的 CLR 存储过程

```
CREATE  PROCEDURE dbo.AddNumber --创建 CLR 存储过程
(
    @a int,
    @b int,
    @c int output --存储过程第 3 个参数为 OUTPUT 类型，与 C#中的 ref 对应
)
--指定 CLR 存储过程对应函数所在的位置
AS EXTERNAL NAME [TestSQLAssembly].[StoredProcedures].[Add]
GO
--以下是调用 CLR 存储过程
DECLARE @c int
EXEC dbo.AddNumber 11,22,@c OUTPUT;
PRINT @c
```

其运行结果为 33。

9.6　创建 CLR 触发器

触发器是数据库服务器中发生事件时自动执行的特种存储过程。如果用户要通过 DML 事件编辑数据，则执行 DML 触发器。DML 事件是针对表或视图的 INSERT、UPDATE 或 DELETE 语句。DDL 触发器用于响应各种 DDL 事件。这些主要是 CREATE、ALTER 和 DROP 语句。通过 T-SQL 语句或使用 Microsoft .NET Framework 公共语言运行时创建的程序集的方法，可以在 SQL Server 中直接创建 DML 和 DDL 触发器，将其上传给一个 SQL Server 实例。本节将主要讲解 CLR 触发器的创建和使用。

9.6.1　使用 C#编写 CLR 触发器

为了能够在 C#中处理触发器触发时的情况，Microsoft.SqlServer.Server 命名空间中提供了 SqlTriggerContext 类，可以通过 SqlContext.TriggerContext 来获得：

```
SqlTriggerContext myTriggerContext = SqlContext.TriggerContext;
```

SqlTriggerContext 类提供了关于触发器的信息，可以通过 TriggerAction 获得触发的类型。在 T-SQL 的触发器中有两张特殊的表 inserted 和 deleted，在使用 C#编写 CLR 触发器中若需要访问这两个表，则需要使用 SqlCommand。例如要访问 inserted 表，则对应的 C#代码如代码 9.24 所示。

代码 9.24　在 C#中访问 inserted 表

```
//将来对当前数据库的连接
SqlConnection connection = new SqlConnection ("context connection = true");
connection.Open();                  //打开连接
SqlCommand command = connection.CreateCommand();
command.CommandText = "SELECT * from " + "inserted";
```

```
reader = command.ExecuteReader();   //执行 SQL 语句
reader.Read();                      //读取数据
//循环每一列
for (int columnNumber = 0; columnNumber < triggContext.ColumnCount;
columnNumber++)
{
  pipe.Send("Updated column "       //将每一列的列名通过 pipe.Send 发送到客户端
     + reader.GetName(columnNumber) + "? "
  + triggContext.IsUpdatedColumn(columnNumber).ToString());
 }
 reader.Close();                    //关闭连接
```

这里要声明的是，context connection = true 表示使用的是当前连接的内容，而不是新建的一个连接。例如现在有两个表：Users 和 Change，Change 表中记录了 Users 中姓张的用户被修改过的次数，也就是说每次更新 Users 表时触发器将统计被修改的姓张的用户，然后将该数据增加到 Change 表中。Users 表和 Change 表的定义如代码 9.25 所示。

代码 9.25　触发器准备的表和数据

```
CREATE TABLE Users --创建用户表
(
    ID int IDENTITY PRIMARY KEY,
    Name nvarchar(20) NOT NULL
)
GO
CREATE TABLE Change --记录用户姓名被修改的次数
(
    Counts int NOT NULL
)
GO
INSERT INTO Users VALUES('张三')
INSERT INTO Users values('李四')
INSERT INTO Change VALUES(0)
```

对于 Users 表要创建更新触发器，则对应的 C#函数如代码 9.26 所示。

代码 9.26　创建的 CLR 触发器

```
using System;
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;
//定义了类 Triggers 作为 CLR 触发器所在的类
public partial class Triggers
{
//通过 Attribute 标明函数 AddChangesTrigger 是用于 CLR 触发器的函数
//属性中定义了目标表是 Users 表，针对的是 UPDATE 操作
    [Microsoft.SqlServer.Server.SqlTrigger    (Name="AddChangesTrigger",
Target="Users", Event="FOR UPDATE")]
    public static void AddChangesTrigger()
    {
        SqlTriggerContext triggContext = SqlContext.TriggerContext;
        if (triggContext.TriggerAction == TriggerAction.Update)
                                                    //如果是更新操作则执行
        {
//将当前连接放入 SqlConnection 对象中
```

```
            using (SqlConnection conn = new SqlConnection("context conne-
            ction=true"))
            {
                conn.Open();                                //打开连接
                SqlCommand command = conn.CreateCommand();
                command.CommandText = "SELECT * from " + "inserted";
                                                            //SQL 语句
                SqlDataReader reader = command.ExecuteReader();
                                                            //执行 SQL 语句
                int count = 0;                              //初始化统计数据
                while (reader.Read())                       //循环遍历数据库中数据
                {
                    string userName = (string)reader[1];
                    if (userName.IndexOf("张") == 0)
                    {
                        count++;                            //统计姓张的被修改的人数
                    }
                }
                reader.Close();
                //以下是更新 Change 表
                SqlCommand sqlComm = conn.CreateCommand();
                                                        //使用当前连接执行 SQL 命令
                SqlPipe sqlP = SqlContext.Pipe;
                sqlComm.CommandText = "UPDATE Change SET Counts=Counts+" +
                count;
                //定义要执行的 SQL 命令
                sqlP.Send(sqlComm.CommandText);
                sqlP.ExecuteAndSend(sqlComm);           //更新 Change 表
            }
        }
    }
}
```

使用前面介绍的同样的方法编译程序为 dll，然后更新 SQL Server 中的程序集。

9.6.2　在 SQL Server 中使用 CLR 触发器

在添加程序集后，将程序集中的触发器函数添加到 SQL Server 中。添加触发器使用 CREATE TRIGGER 命令。创建 CLR 触发器的语法如代码 9.27 所示。

代码 9.27　创建 CLR 触发器的语法

```
CREATE TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS EXTERNAL NAME assembly_name.class_name.method_name
```

其语法与创建 T-SQL 触发器基本相同，assembly_name.class_name.method_name 用于指定程序集与触发器绑定的方法。该方法不能带有任何参数，并且必须返回空值。class_name 必须是有效的 SQL Server 标识符，并且该类必须存在于可见程序集中。

现将 TestSQLAssembly 程序集中的 AddChangesTrigger 触发器添加到 SQL Server，脚

本如代码 9.28 所示。

代码 9.28　创建 CLR 触发器

```
CREATE TRIGGER AddChangesTrigger --创建 CLR 触发器
ON Users FOR UPDATE --触发器针对的表和操作
--以下定义 CLR 触发器所在的位置
AS EXTERNAL NAME [TestSQLAssembly].[Triggers].AddChangesTrigger
```

添加触发器后便可更改 Users 表中的数据，看姓张的用户变动是否引起了 Change 表中数值的变化。在 SQL Server 中测试该 CLR 触发器的脚本如代码 9.29 所示。

代码 9.29　测试 CLR 触发器

```
SELECT * FROM Change --初始是 0
GO
UPDATE Users SET Name='张小二'
WHERE Name LIKE '张%'
GO
--修改了 Users 表数据以后
SELECT * FROM Change --现在是 1
```

9.7　创建用户定义聚合函数

在 SQL Server 中，经常需要对数据按组进行自定义的聚合操作，但默认只有 SUM()、MAX()、MIN()和 AVG()等聚合函数。在 SQL Server 2012 中提供了编写 CLR 的托管代码的支持，可以用来创建自定义的聚合函数。

9.7.1　使用 C#编写聚合函数

在 SqlServerProject1 项目中添加用户定义聚合函数，系统将自动创建聚合函数的模板，如代码 9.30 所示。

代码 9.30　用户定义聚合函数模板

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)]
public struct Aggregate1
{
    public void Init()
    {
        //输入你的代码
    }
    public void Accumulate(SqlString Value)
    {
```

```
        //输入你的代码
    }
    public void Merge(Aggregate1 Group)
    {
        //输入你的代码
    }
    public SqlString Terminate()
    {
        //输入你的代码
        return new SqlString("");
    }
    // This is a place-holder member field
    private int var1;
}
```

从模板中可以看出，用于创建聚合函数的类除了标记为 SqlUserDefinedAggregate 外还必须是可序列化的。类中必须提供 4 个方法：Init()、Accumulate()、Merge()和 Terminate()，这 4 个方法的含义分别是初始化、扫描到一条记录时、合并、终止扫描。创建聚合函数时只需要将要实现的功能写入这 4 个函数即可。

例如要创建对元音字母（a、e、i、o、u）进行计数的聚合函数，则对应的 C#代码如代码 9.31 所示。

代码 9.31　创建元音字母计数聚合函数

```
using System;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text;
//以下 Attribute 标明结构体 CountVowels 是可序列化的，而且被用于用户定义 CLR 聚合函数
[Serializable]
[SqlUserDefinedAggregate(Format.Native)]
//定义结构体 CountVowels
public struct CountVowels
{
    private int countOfVowels;
    public void Init()                              //初始化函数
    {
        countOfVowels = 0;
    }
    //以下函数定义了聚合函数的具体聚合操作
    public void Accumulate(SqlString value)
    {
        char[] vowels = "aeiou".ToCharArray();  //获得元音数组
        char[] strs = value.Value.ToLower().ToCharArray();
                                    //将传入的参数转换为小写的字符数组
        for (int i = 0; i < strs.Length; i++)
                                    //对于传入参数字符数组的每一个字符循环操作
        {
            for (int j = 0; j < vowels.Length; j++)
                                    //对每一个元音字母都进行匹配
            {
                if (strs[i] == vowels[j])           //如果匹配，则元音总数加 1
                {
                    countOfVowels += 1;
                }
            }
```

```
        }
    }
    //定义合并函数，将元音数相加
    public void Merge(CountVowels value)
    {
        countOfVowels += value.countOfVowels;
    }
    //定义结束函数，返回元音总数值
    public SqlInt32 Terminate()
    {
        return new SqlInt32( countOfVowels);
    }
}
```

编写好聚合函数后，重新编译整个项目然后使用 ALTER ASSEMBLY 命令将聚合函数添加到 SQL Server 的程序集中。

9.7.2　在 SQL Server 中创建用户定义聚合函数

在添加了程序集后，要在 SQL Server 中创建用户定义聚合函数以引用 CLR 中的聚合函数。创建用户定义聚合函数使用 CREATE AGGREGATE 命令，其语法如代码 9.32 所示。

代码 9.32　创建聚合函数的语法

```
CREATE AGGREGATE [ schema name . ] aggregate name
        (@param name <input sqltype>
        [ ,...n ] )
RETURNS <return sqltype>
EXTERNAL NAME assembly name [ .class name ]
<input_sqltype> ::=
       system_scalar_type | { [ udt_schema_name. ] udt_type_name }
<return sqltype> ::=
       system_scalar_type | { [ udt_schema_name. ] udt_type_name }
```

其中各参数的含义如下所述。

- schema_name：用户定义聚合函数所属的架构的名称。
- aggregate_name：要创建的聚合函数的名称。
- @param_name：用户定义聚合函数中的一个或多个参数。在执行聚合函数时，用户必须提供参数的值。通过将“at”符号（@）用做第一个字符来指定参数名称。参数名称必须符合标识符规则。该函数的参数是局部参数。
- system_scalar_type：要存放输入参数值或返回值的任意一个 SQL Server 系统标量数据类型。除 text、ntext 和 image 之外的所有标量数据类型，都可用做用户定义聚合函数的参数。不能指定非标量类型（如 cursor 和 table）。
- udt_schema_name：CLR 用户定义类型所属的架构的名称。如果未指定，则数据库引擎将按以下顺序引用 udt_type_name：本机 SQL 类型命名空间、当前数据库中当前用户的默认架构、当前数据库中的 dbo 架构。
- udt_type_name：当前数据库中已创建的 CLR 用户定义类型的名称。如果未指定 udt_schema_name，则 SQL Server 假定该类型属于当前用户的架构。

- assembly_name[.class_name]：指定与用户定义的聚合函数绑定在一起的程序集，以及（可选）该程序集所属的架构名称和该程序集中实现该用户定义聚合函数的类名称。必须先使用 CREATE ASSEMBLY 语句在数据库中创建该程序集。class_name 必须是有效的 SQL Server 标识符并与该程序集中现有的类名称相匹配。如果编写类所用的编程语言使用了命名空间（如 C#），则 class_name 可以是命名空间限定名称。如果未指定 class_name，则 SQL Server 假定该名称与 aggregate_name 相同。

注意：与创建 CLR 函数、CLR 存储过程和 CLR 触发器不同，聚合函数中只使用 assembly_name.class_name 来引用而无法定义 method_name，因为聚合函数对应的是一个 struct 类型而不是一个方法。

使用 CREATE ASSEMBLY 命令，创建用于统计元音字母的聚合函数的脚本，如代码 9.33 所示。

代码 9.33　创建用户定义聚合函数

```
CREATE AGGREGATE CountVowels(@input nvarchar(4000)) --创建用户定义聚合函数
RETURNS int
EXTERNAL NAME [TestSQLAssembly].CountVowels --指定聚合函数在程序集中的位置
```

创建好 CountVowels 聚合函数后便可在 T-SQL 中使用该聚合函数。例如要查询出每个城市的人员数和总城市名中的元音字母的个数，则对应的脚本如代码 9.34 所示。

代码 9.34　使用用户定义聚合函数

```
--使用用户定义聚合函数查询数据
SELECT   City,count(City)   AS   PersonCount,dbo.CountVowels(City)   AS
CityVowelsCount
FROM Person.Address
GROUP BY City
```

9.8　创建 CLR 用户定义类型

通过定义供 SQL Server 编程使用的自定义数据类型来扩展 SQL 类型系统。用户定义类型（UDT）可以是简单的，也可以是结构化的，并且复杂程度可以是任意的。在 UDT 中可以封装复杂的、用户定义的属性和方法。用户定义类型可用于定义表中列的类型，或者 T-SQL 语言中的变量或例程参数的类型。用户定义类型实例可以是表中的列，批处理、函数或存储过程中的变量，或者函数或存储过程的参数。

9.8.1　使用 C#定义类型

与用户定义聚合函数的要求类似，用户定义类型也是在 C#中用一个可序列化的结构体表示。用户定义类型必须满足接口 INullable，声明 IsNull 属性表示该类型是否为空值。在用户定义类型中可以声明其他的属性和方法，这些属性和方法在 SQL Server 中可以被调用。

假设需要创建一个点类型 Point 用于表示二维空间中的一个点，通过该类型可获得其 X 坐标、Y 坐标和该点距原点的距离。在项目中添加 Point 用户定义类型，其 C#代码如代码 9.35 所示。

代码 9.35　使用 C#创建 Point 类型

```
using System;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
//以下 Attribute 标明结构体 Point 是可序列化的，作为用户定义数据类型使用
[Serializable]
[SqlUserDefinedType(Format.Native)]
public struct Point : INullable //用户定义类型必须满足接口 INullable
{
//以下是定义结构体的内部变量和属性
    private Int32 x;
    private Int32 y;
    private bool is_Null;
    public Int32 X //X 坐标
    {
        get        {            return (this.x);        }
        set        {            x = value;        }
    }
    public Int32 Y //Y 坐标
    {
        get        {            return (this.y);        }
        set        {            y = value;        }
    }
    public bool IsNull                          //是否为空
    {
        get        {            return is_Null;        }
    }
    public static Point Null                    //定义一个空对象属性
    {
        get
        {
            Point pt = new Point();
            pt.is_Null = true;
            return pt;
        }
    }
    //重写 ToString 方法，指明在以字符串形式输出该类型时输出的内容
    public override string ToString()
    {
        if (this.IsNull)
        {
            return "NULL";                      //输出 NULL 字符串
        }
        else
        {
            return this.x + "," + this.y;     //输出 x，y 的形式表示一个二维点
        }
    }
    public double Len()                     //定义函数 Len，用于返回该点到原点的距离
    {
        if (IsNull)
        {
            return 0;
```

```
        }
        return Math.Sqrt(x * x + y * y);
    }
//定义静态函数 Parse，指明如何将字符串转换为该用户定义的数据类型
    public static Point Parse(SqlString s)
    {
        if (s.IsNull)                        //如果传入为空，则返回空
        {
            return Null;
        }
        string str = s.Value;
        string[] xy = str.Split(',');
        Point pt = new Point();
        pt.X = Convert.ToInt32(xy[0]);
        pt.Y = Convert.ToInt32(xy[1]);
        return pt;                           //返回 Point 对象
    }
}
```

创建好用户定义类型后编译该项目，然后将生成的 dll 文件更新到 SQL Server 程序集中。

9.8.2　在 SQL Server 中使用 CLR 用户定义类型

在 SQL Server 中创建 CLR 用户定义类型使用 CREATE TYPE 命令。CREATE TYPE 可以创建基于 SQL 数据类型的用户定义类型，也可以创建基于 CLR 的用户定义类型。创建用户定义类型的语法，如代码 9.36 所示。

代码 9.36　CREATE TYPE 语法

```
CREATE TYPE [ schema name. ] type name
  EXTERNAL NAME
  assembly_name [ .class_name ]
```

其中每个参数的含义如下所述。

- ❑ schema_name：标明数据类型或用户定义类型所属架构的名称。
- ❑ type_name：标明数据类型或用户定义类型的名称。类型名称必须符合标识符的规则。
- ❑ assembly_name：指定可在公共语言运行库中引用用户定义类型实现的 SQL Server 程序集。assembly_name 应与当前数据库的 SQL Server 中的现有程序集匹配。
- ❑ [.class_name]：指定实现用户定义类型的程序集内的类。class_name 必须是有效的标识符，而且必须在具有程序集可见性的程序集中作为类存在。class_name 区分大小写，也不管数据库的排序规则如何，且必须与对应的程序集中的类名完全匹配。如果用于编写类的编程语言使用命名空间概念（例如 C#），则类名可以是用方括号（[]）括起的限定命名空间的名称。如果未指定 class_name，SQL Server 假定该名称与 type_name 相同。

这里要创建 Point 类型，引用 CLR 中的 Point 类型，则对应的创建 Point 用户定义类型的 SQL 脚本如代码 9.37 所示。

代码 9.37　创建 CLR 用户定义类型

```
CREATE TYPE Point --创建用户定义数据类型
EXTERNAL NAME [TestSQLAssembly].Point --指定该数据类型在程序集中的位置
--或者可以直接简写为:
CREATE TYPE Point
EXTERNAL NAME [TestSQLAssembly]
```

创建 Point 类型后便可在列类型、变量和参数中使用，例如创建基于 Point 类型的表，然后插入、查询其中的数据。具体示例如代码 9.38 所示。

代码 9.38　使用 CLR 用户定义类型

```
CREATE TABLE testPoint --创建一个测试表
(
    c1 Point--使用 CLR 用户定义数据类型作为 c1 的数据类型
)
GO
INSERT INTO testPoint VALUES('12,2') --插入一些值
INSERT INTO testPoint VALUES('12,4')
INSERT INTO testPoint VALUES('3,4')
GO
SELECT c1.X AS X,c1.Y AS Y,c1.Len() AS LEN --调用 CLR 数据类型定义的属性和方法
FROM testPoint
```

9.9　小　　结

本章主要讲解了 SQL Server 与 CLR 的集成使用。对 CLR 集成的支持是 SQL Server 2005 中添加的新特性。SQL Server 2012 在支持原有 CLR 集成的特性外，还扩大了 CLR 类型所允许的空间大小，用户定义类型和用户定义聚合函数限制最大到 8000 字节，而在 SQL Server 2012 中扩大到 2GB。

SQL Server 与 CLR 集成能够在 SQL Server 中创建 CLR 函数、CLR 存储过程、CLR 触发器、用户定义聚合函数和 CLR 用户定义类型。通过与 CLR 的集成，使 SQL Server 的功能可以进行无限扩展，复制的逻辑算法（比如加密算法）等可以在 SQL Server 中轻松实现。

第 10 章　在 SQL Server 中使用 XML

XML 本身并不是作为一种数据库技术而设计的，但是由于 XML 的自描述性、可扩展能力和跨平台优势已经成为各种异构系统之间通信的主要方式，所以各大数据库厂商都在其产品中增加了对 XML 的支持。从 SQL Server 2005 开始，SQL Server 为 XML 数据处理提供了广泛支持，SQL Server 2008 在 XML 处理上并没有添加新功能，所以本章学习的知识在 SQL Server 2005 中同样适用。本章将主要学习 XML 的相关知识和 SQL Server 2012 对 XML 的支持和操作。

10.1　XML 概述

本节将主要介绍 XML 的基本概念及对 XML 数据的操作，了解这些基础知识才能更好地学习和理解 SQL Server 2012 对 XML 的支持和操作。

10.1.1　XML 简介

XML 是可扩展标记语言（Extensible Markup Language）的简写，最初是为文档管理而设计的，其前身是 SGML（Standard Generalized Markup Language），在 1998 年 2 月发布了 W3C（Word Wide Web Concortium 的缩写，W3C 组织是对网络标准制定的一个非营利组织）的标准。

对于包括 XML、HTML（Hyper Text Markup Language）在内的标记语言家族，标记都是采用封闭的尖括号（< >）中的标签。标签都是成对地使用，以<tag>和</tag>来确定标记的开始和结束。例如，在 HTML 中经常可以看到这样的标记：

```
<title>首页</title>
```

与 HTML 不同的是，XML 中没有指定具体的标签集及标签的含义，每个应用都可以自己定义需要的标签集和标签含义。这一个特性使 XML 主要用于数据表示和数据交换中；而 HTML 是应用于文档格式化的关键所在。如代码 10.1 就表示了一个公司的组织结构，是对 XML 的一个更加现实的应用。

代码 10.1　XML 文档格式

```
<company>
    <department>
        <departmentName>ADC</departmentName>
        <departmentID>12</departmentID>
        <leader>Shing</leader>
```

```
    </department>
    <department>
        <departmentName>FA</departmentName>
        <departmentID>13</departmentID>
        <leader>Zhang</leader>
        <employeeCount>15</employeeCount>
        <groups>
            <group>Dev</group>
            <group>QA</group>
            <group>Pro</group>
        </groups>
    </department>
</company>
```

与关系数据库中存储的数据相比，XML 表达式的效率可能并不高，因为标签名称在整个文档中被反复使用。虽然有这些缺点，但是 XML 表达式在异构系统的数据交换及文件中存储结构化信息有巨大的优势：

- XML 的标签是自描述的，不需要参考其他的文档就可以理解 XML 文档的含义。不需要任何文档从字面上就可以看出代码 10.1 中定义了一个公司，该公司下面有两个部门，同时也可以知道部门的编号、部门名、部门经理等信息。
- XML 对格式没有严格限制。代码 10.1 中的 XML 文档提供了两个部门信息，ADC 部门只提供了部门名、部门编号、部门经理；而 FA 部门另外还提供了雇员数信息和分组信息。XML 识别程序既可以从中提取出雇员数信息或分组信息也可以直接忽略这个信息。这种识别和忽略标签的能力使得 XML 数据格式可以随着需求不断变化而不需舍弃原来的应用程序。
- XML 允许嵌套结构。代码 10.1 中的 XML 文档在公司标签下嵌套了部门标签，部门标签下嵌套部门名标签、分组标签等，分组标签下再继续嵌套组标签。这样的信息如果是在关系数据库中将会被拆分成多个关系以帮助避免冗余，而在 XML 文档中则可以通过嵌套结构来表示，使交换信息更易读。

XML 格式作为 W3C 标准，已经被广泛接受，有各种各样的工具软件来辅助对 XML 的处理。

10.1.2　XML 数据的结构

XML 文档中最基本的结构是元素（element）。一个元素就是一对匹配的标签及其中的文本。XML 中的元素必须正确嵌套。比如：

```
<company>…<department>…</department>…</company>
```

就是正确的嵌套，而

```
<company>…<department>…</company>…</department>
```

就是错误的嵌套。另外，如果一个元素形如<ele></ele>，其中不包含任何子元素或文本，那么可以缩写成<ele/>。

XML 文档是一个树结构文档，就像目录结构所有文件都必须存在一个盘符的目录下一样，XML 文档中必须有一个独立的根元素来包含文档中所有的其他元素。在代码 10.1 中<company>就是根元素。

除了元素之外，XML 还定义了属性（attribute）。一个元素的属性是以标签结束符“>”之前的 name=value 形式出现的。属性是字符串，不包含标记，而且属性在给定标签中只可以出现一次，不像子元素那样可以重复。比如：

```
<user name="ZY" age="23">Zeng</user>
```

由于 XML 文档被用于应用程序之间的数据交换，人们引入了名字空间（namespace）机制以允许一些组织机构指定全球唯一的名字作为文档中的标签来使用。名字空间的写法就是在每个标签或属性前面加上通用资源标识符（比如网址）。为了书写和阅读的方便，可以为名字空间使用一种缩写的方法。比如代码 10.2 中，在根元素<company>中有个属性 XMLns:ABC，用于声明 ABC 为给定 URL 的缩写，这个缩写可以用于各种元素标签中。

代码 10.2　名字空间的使用

```
<company XMLns:ABC=http://www.aabbcc.com>
    ...
    <ABC:department>
        <ABC:departmentName>ADC</ABC:departmentName>
        ...
    </ABC:department>
    ...
</company>
```

另外，可以使用属性 XMLns 代替 XMLns:ABC 定义默认的名字空间。所有没有显式给出名字空间前缀的元素都属于默认的名字空间。

如果需要存储包含标签的值而不希望将其中的标签解释为 XML 标签，那么可以使用“<![CDATA[“和”]]>”将数据包含起来。CDATA 是字符数据的意思，表示其中的所有内容都被当做一般字符串来处理，而不会被当做 XML 标签。比如：

```
<info><![CDATA[<title>首页</title>]]></info>
```

其中的<title>首页</title>将被解释为<info>标签中的数据，而不会被解释为<info>的子标签。

10.1.3　XML 文档模式

在关系数据库中可以规定一个字段的数据类型、是否允许为空、有何限制等，但是 XML 文档中的元素却可以被任意地创建、嵌套。当需要使用应用程序对 XML 进行自动处理时，统一的 XML 格式和统一的含义是很有必要的，于是出现了 XML 文档模式。

第一种模式定义语言：文档类型定义（document type definition，DTD）能够限制并归类 XML 中的信息。但是 DTD 只能限制元素中的子元素和属性的出现，却不能限制整数字符串以外的复杂类型，加上 DTD 自身的一些其他缺陷，使得 DTD 在现实应用中已经很少使用了，取而代之的是更完善的模式语言 XML 架构（XML Schema），所以笔者在此不对 DTD 作过多介绍，主要介绍 XML Schema。

XML Schema 定义了很多内部类型，比如 string、integer、decimal、date 和 boolean。另外，XML Schema 还允许用户定义类型，这些用户定义类型可能是增加了限制的简单类型，或者使用构造器 complexType 和 sequence 构造的复制类型。假如有一个用户基本信息

的 XML 文档，其内容如代码 10.3 所示。

代码 10.3　用户基本信息的 XML 文档

```
<users>
    <user>
        <name>张三</name>
        <sex>男</sex>
        <age>22</age>
    </user>
    <user>
        <name>李四</name>
        <sex>女</sex>
        <age>21</age>
    </user>
</users>
```

那么其 XML Schema 如代码 10.4 所示。

代码 10.4　用户基本信息的 XML Schema 定义

```
<xs:schema XMLns:xs="http://www.w3.org/2001/XMLSchema" elementFormDef-
ault="qualified">
    <xs:element name="age" type="xs:unsignedInt"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="sex">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="男"/>
                <xs:enumeration value="女"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
    <xs:element name="user">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="name"/>
                <xs:element ref="sex"/>
                <xs:element ref="age"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="users">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="user" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

需要注意的是，XML Schema 中的模式定义也是使用 XML 预先指定的，为了避免与用户标签发生冲突，XML Schema 标签加上了前缀“xs:”并指定了其名字空间。

从 XML Schema 文档的最后向上分析：

```
<xs:element name="users">
    ...
</xs:element>
```

该段代码表示根元素 users 是一个复杂类型，其子元素为 user，ref 表明 user 元素在前面的 XML Schema 中已经定义。XML Schema 可以使用 minOccurs 和 maxOccurs 来指定某个元素出现的最少次数和最多次数。最少次数和最多次数的默认值都是 1，unbounded 表明没有做次数限制。

从 user 元素的 XML Schema 定义中可以得到：user 元素也是一个复杂类型，其内容是由 name、sex、age 这 3 个元素组成的，而且由于没有指定 minOccurs 和 maxOccurs，所以这 3 个元素在 user 元素中都只出现了一次。同样，ref 表明这些元素在前面的 XML Schema 中已经定义。

而 sex 元素则是被定义为一个枚举类型，只允许“男”或“女”作为其内容。name 被定义为字符串类型，允许任意字符串作为其内容。显然年龄是不会出现负值的，所以 age 被定义为无符号整型，即 age 为正整数。

XML Schema 有很多优于 DTD 之处，目前已经得到广泛应用。其主要特点有：

- ❑ 允许把元素中出现的文本限制为专门的类型或复杂类型。
- ❑ 允许创建自定义类型。
- ❑ 允许唯一性和外键约束。
- ❑ 与名字空间结合，可以实现 XML 文档中的不同部分遵循不同的模式。
- ❑ 允许使用一种形式的继承来扩展复杂类型。

在此笔者只是对 XML 进行简单概括性的介绍，如果读者想进一步了解 XML 的相关信息，可以登录 http://www.w3c.org 或查看相关书籍资料。

10.2　FOR XML 子句的模式

FOR XML 子句主要用于跟在查询语句之后，使 SQL 查询将结果返回为 XML，而不是标准行集。在 FOR XML 子句中，必须指定 XML 模式为 AUTO、RAW、PATH 或 EXPLICIT。这 4 种模式的特点是：

- ❑ RAW 模式将为 SELECT 语句所返回行集中的每行生成一个<row>元素。可以通过编写嵌套 FORXML 查询生成 XML 层次结构。
- ❑ AUTO 模式将基于指定 SELECT 语句的方式，使用试探性方法在 XML 结果中生成嵌套。用户对生成的 XML 的形状具有最低限度的控制能力。除了 AUTO 模式的试探性方法生成的 XML 形状之外，还可以编写 FORXML 查询生成 XML 层次结构。
- ❑ EXPLICIT 模式允许对 XML 的形状进行更多控制。可以随意混合属性和元素来确定 XML 的形状。由于执行查询而生成的结果行集需要具有特定的格式，所以此行集格式随后将映射为 XML 形状。使用 EXPLICIT 模式能够随意混合属性和元素、创建包装和嵌套的复杂属性、创建用空格分隔的值（例如 OrderID 属性可能具有一列排序顺序 ID 值）及混合内容。
- ❑ PATH 模式与嵌套 FORXML 查询功能一起，以较简单的方式提供了 EXPLICIT 模式的灵活性。

10.2.1　RAW 模式

RAW 模式将查询结果集中的每一行转换为带有通用标识符<row>或可能提供元素名称的 XML 元素。默认情况下，行集中非 NULL 的每列值都将映射为<row>元素的一个属性。如果将 ELEMENTS 指令添加到 FORXML 子句中，则每个列值都将映射到<row>元素的子元素。指定 ELEMENTS 指令之后，还可以选择性地指定 XSINIL 选项以将结果集中的 NULL 列值映射到具有 xsi:nil="true"属性的元素。

可以请求返回所产生的 XML 的架构。指定 XMLDATA 选项将返回内联 XDR 架构。指定 XMLSCHEMA 选项将返回内联 XSD 架构。该架构显示在数据的开头。在结果中，每个顶级元素都引用架构命名空间。

必须在 FORXML 子句中指定 BINARYBASE64 选项，以使用 base64 编码格式返回二进制数据。在 RAW 模式下，如果不指定 BINARYBASE64 选项就检索二进制数据，将导致错误。以 AdventureWorks 数据库为例，若要以 RAW 模式 XML 格式返回客户的地址和客户名字，则对应的 SQL 脚本如代码 10.5 所示。

代码 10.5　RAW 模式返回 XML 代码

```
SELECT TOP 2 pe.EmailAddress,pp.FirstName,pp.LastName
FROM Person.Person pp
INNER JOIN Person.EmailAddress pe
ON pp.BusinessEntityID = pe.BusinessEntityID
FOR XML RAW --使用 RAW 模式
```

返回结果为：

```
<row EmailAddress="ken0@adventure-works.com" FirstName="Ken" LastName="Sánchez" />
<row EmailAddress="terri0@adventure-works.com" FirstName="Terri" LastName="Duffy" />
```

这里为了便于查看 XML 结果，所以只返回了 2 行记录。从结果中可以看到，针对每一行数据，XML 中产生了一个 row 元素，所有列的信息无论是来自哪张表都作为元素的属性统一显示在 row 元素中。这样做的缺点是不能从结果中反映出真实的层次特性；其优点是涉及 XML DOM 的内容不是很深，占用的内存较小，执行效果好。

除了使用默认的 row 元素名外还可以为生成的元素命名，例如若将 row 元素改为 Address 元素，则对应的 SQL 脚本和返回的结果如代码 10.6 所示。

代码 10.6　修改 row 元素名

```
SELECT TOP 2 at.AddressTypeID,at.Name,ca.CustomerID,cabe.ModifiedDate,be.AddressID
FROM Person.AddressType at
INNER JOIN Sales.CustomerAddress ca
INNER JOIN Person.BusinessEntityAddress be
ON cabe.AddressTypeID = at.AddressTypeID
FOR XML RAW('Address')   --指定了 row 元素名
```

其返回结果为：

```
<Address  AddressTypeID="2"  Name="Home"  CustomerID="11000ModifiedDate=
"2008-10-13T11:15:06.967" AddressID="22601249" />
<Address  AddressTypeID="2"  Name="Home"  CustomerID="11001ModifiedDate=
"2008-10-13T11:15:06.967" AddressID="14489293" />
```

从结果中可以看出，RAW 模式将查询结果集中的每一行转换为提供元素名称的 XML 元素属性。如果没有提供参数“'Address'”，那么每行将使用通用标识符<row>表示。

这里需要注意的是，SQL Server Management Studio 运行 SQL 语句以后返回的是一行数据集，其界面如图 10.1 所示。若要查看返回的完整 XML 内容，可以单击返回结果中的超链接。

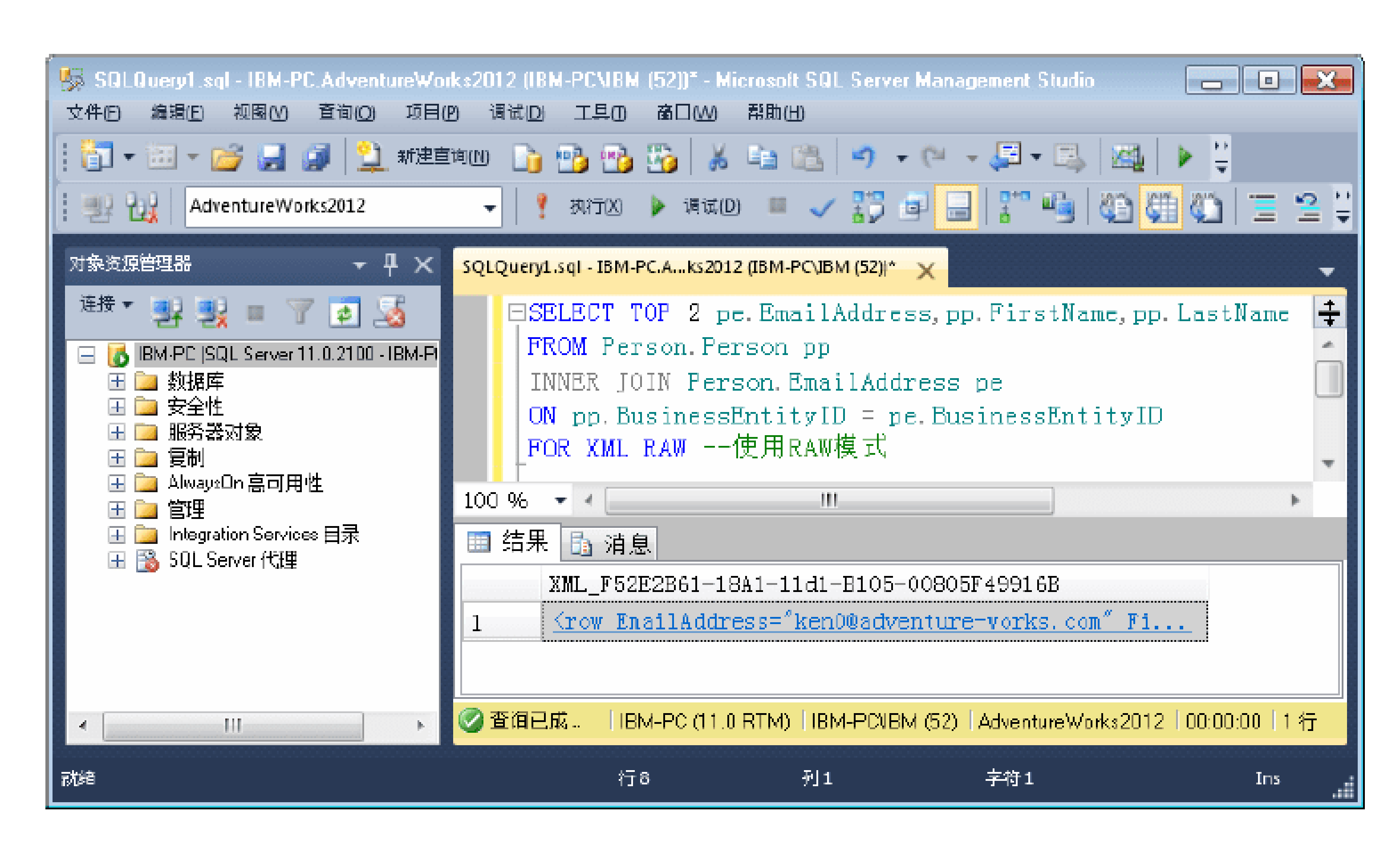


图 10.1　SSMS 执行 SQL 语句

10.2.2　AUTO 模式

AUTO 模式处理数据的方法与 RAW 不同，AUTO 模式将查询结果以嵌套 XML 元素的方式返回。AUTO 模式基于表名（或者表的别名）命名元素，在 XML 中提供的数据可能具有一些基础层次结构。以 AdventureWorks 示例数据库内容为例，运行代码 10.7 可得到返回的 XML 结果。

代码 10.7　使用 AUTO 模式的 FOR XML 子句

```
SELECT TOP 2 pe.EmailAddress,pp.FirstName,pp.LastName
FROM Person.Person pp
INNER JOIN Person.EmailAddress pe
ON pp.BusinessEntityID = pe.BusinessEntityID
FOR XML AUTO --使用 FOR XML 子句使返回结果为 XML 类型
```

其输出结果为：

```
<pe EmailAddress="ken0@adventure-works.com">
  <pp FirstName="Ken" LastName="Sánchez" />
</pe>
<pe EmailAddress="terri0@adventure-works.com">
  <pp FirstName="Terri" LastName="Duffy" />
</pe>
```

FOR XML 中的 AUTO 模式表示自动检测表的链接关系，并使用嵌套元素的 XML 表示方式返回结果。也就是说，从 XML 结果的嵌套关系可以看出，查询是 at 表和 ca 表的链接查询。之所以 at 元素是 ca 元素的父级元素，是因为在 SELECT 中 at 表排在前面，若将 ca 表放在第一个，则输出结果中 ca 元素便成了 at 元素的父级元素，如代码 10.8 所示。

代码 10.8　交换位置的 FOR XML 查询

```
SELECT TOP 2 pp.FirstName,pp.LastName,pe.EmailAddress
FROM Person.Person pp
INNER JOIN Person.EmailAddress pe
ON pp.BusinessEntityID = pe.BusinessEntityID
FOR XML AUTO    --使用 AUTO 模式
```

其输出结果为：

```
<pp FirstName="Ken" LastName="Sánchez">
  <pe EmailAddress="ken0@adventure-works.com" />
</pp>
<pp FirstName="Terri" LastName="Duffy">
  <pe EmailAddress="terri0@adventure-works.com" />
</pp>
```

10.2.3　EXPLICIT 模式

使用 RAW 和 AUTO 模式不能很好地控制查询结果生成的 XML 的形状。但是，对于要从查询结果生成 XML，EXPLICIT 模式会提供非常好的灵活性。必须以特定的方式编写 EXPLICIT 模式查询，以便将所需的 XML 附加信息（如 XML 中的所需嵌套）显式指定为查询的一部分。根据所请求的 XML，编写 EXPLICIT 模式查询可能会很繁琐，而使用 PATH 模式（具有嵌套）相对编写 EXPLICIT 模式查询而言更加简单。

因为要将所需的 XML 描述为 EXPLICIT 模式查询的一部分，所以必须确保生成的 XML 格式正确且有效。为使 EXPLICIT 模式生成 XML 文档，行集必须具有特定的格式。这就需要编写 SELECT 查询以生成具有特定格式的行集（通用表），以便生成所需的 XML。

1. 通用表

要使用 EXPLICIT 模式，则查询必须生成下列两个元数据列：

- 第一列必须提供当前元素的标记号（整数类型），并且列名必须是 Tag。查询必须为行集构造的每个元素提供唯一标记号。
- 第二列必须提供父元素的标记号，并且此列的列名必须是 Parent。这样，Tag 和 Parent 列将提供层次结构信息。

这些元数据列值与列名中的信息一起都用于生成所需的 XML。请注意，查询必须以特定的方式提供列名。同时，Parent 列中的 0 或 NULL 表明相应的元素没有父级。该元素将作为顶级元素添加到 XML 中。

为了了解如何将查询生成的通用表处理为生成的 XML 结果，如图 10.2 给出了一个通用表的示例。

Tag	Parent	Customer!1!cid	Customer!1!name	Order!2!id	Order!2!date	OrderDetail!3!id!id	OrderDetail!3!pid!idref
1	NULL	C1	"Janine"	NULL	NULL	NULL	NULL
2	1	C1	NULL	01	1/20/1996	NULL	NULL
3	2	C1	NULL	01	NULL	OD1	P1
3	2	C1	NULL	01	NULL	OD2	P2
2	1	C1	NULL	02	3/29/1997	NULL	NULL

图 10.2　EXPLICIT 模式使用的通用表

关于通用表的格式，有以下注意事项：

- 前两列是 Tag 和 Parent，它们是元数据列。这些值确定层次结构。
- 列名是以特定的方式指定的。
- 从此通用表生成 XML 的过程中，此表中的数据被垂直分区到列组中。分组是根据 Tag 值和列名确定的。在构造 XML 的过程中，处理逻辑为每行选择一组列，然后构造一个元素。在此示例中，将应用以下规则：
  - 对于第一行中的 Tag 列值 1，名称中包括此相同标记号的列（Customer!1!cid 和 Customer!1!name）形成一组。这些列用于处理行，读者可能已经注意到所生成元素的形式为<Customer id=... name=...>。
  - 对于 Tag 列值为 2 的行，列 Order!2!id 和 Order!2!date 形成一组，然后该组用于构造元素（形式为 <Order id=... date=... />）。
  - 对于 Tag 列值为 3 的行，列 OrderDetail!3!id!id 和 OrderDetail!3!pid!idref 形成一组。其中每一行都从这些列生成一个元素（形式为<OrderDetail id=... pid=...>）。

在生成 XML 层次结构的过程中，行是按顺序处理的，Tag 和 Parent 元数据列中的值、列名中提供的信息及正确的行顺序将生成所需的 XML。对于图 10.2 中所示的通用表，生成 XML 文档的过程如下。

（1）第一行指定 Tag 值为 1，Parent 值为 NULL。因此，相应的元素（即<Customer> 元素）将作为顶级元素添加在 XML 中。

```
<Customer cid="C1" name="Janine">
```

（2）第二行确定 Tag 值为 2，Parent 值为 1。因此，<Order>元素将作为<Customer>元素的子元素被添加。

```
<Customer cid="C1" name="Janine">
  <Order id="O1" date="1/20/1996">
```

（3）下两行确定 Tag 值为 3，Parent 值为 2。因此，两个<OrderDetail>元素将作为<Order>元素的子元素被添加。

```
<Customer cid="C1" name="Janine">
  <Order id="O1" date="1/20/1996">
    <OrderDetail id="OD1" pid="P1"/>
    <OrderDetail id="OD2" pid="P2"/>
```

（4）最后一行确定 Tag 号为 2，Parent 标记号为 1。因此，另一个<Order>元素将作为<Customer>父元素的子元素被添加。最后生成的 XML 文档如代码 10.9 所示。

代码 10.9　使用 EXPLICIT 生成的 XML 文档

```
<Customer cid="C1" name="Janine">
  <Order id="O1" date="1/20/1996">
    <OrderDetail id="OD1" pid="P1"/>
    <OrderDetail id="OD2" pid="P2"/>
  </Order>
  <Order id="O2" date="3/29/1997">
</Customer>
```

2. 列名格式

虽然 Tag 和 Parent 都具有整齐的划分点，但是名字占用了许多元数据，区分一个截止点和下一个开始点的唯一方法是感叹号（!）。在编写 EXPLICIT 模式查询时，必须使用以下格式指定所得到的行集中的列名。它们提供转换信息（包括元素名称和属性名称）及用指令指定的其他附加信息。

```
ElementName!TagNumber!AttributeName!Directive
```

下面是对格式各部分的说明。

- ElementName：所生成元素的通用标识符。例如，如果将 ElementName 指定为 Customers，将生成<Customers>元素。
- TagNumber：分配给元素的唯一标记值。在两个元数据列（Tag 和 Parent）的帮助下，此值将确定所得到的 XML 中的元素嵌套。
- AttributeName：提供要在指定的 ElementName 中构造的属性名称。如果没有指定 Directive，将发生这种行为。
- 如果指定了 Directive 并且它是 XML、cdata 或 element，则此值用于构造 ElementName 的子元素，并且此列值将添加到该子元素。
- 如果指定了 Directive，则 AttributeName 可以为空。例如 ElementName! TagNumber!! Directive。在这种情况下，列值直接由 ElementName 包含。
- Directive：Directive 是可选的，可以使用它来提供有关 XML 构造的其他信息。

Directive 有两种用途。一种用途是将值编码为 ID、IDREF 和 IDREFS。可以将 ID、IDREF 和 IDREFS 关键字指定为 Directives，这些指令将覆盖属性类型，这样能够创建文档内链接。

同时，可以使用 Directive 指示如何将字符串数据映射到 XML。可以将 hide、element、elementxsinil、XML、XMLtext 和 cdata 关键字用做 Directive。hide 指令会隐藏节点，当仅为排序目的而检索值，但又不想让它们出现在所得到的 XML 中时，此指令非常有用。

element 指令生成的结果中包含元素而不是属性。包含的数据被编码为实体。例如，<字符变成<。对于 NULL 列值，不会生成任何元素。如果要为 NULL 列值生成元素，可以指定 elementxsinil 指令。这将生成具有属性 xsi:nil=TRUE 的元素。

除不发生实体编码外，XML 指令与 element 指令相同。请注意，可以将 element 指令与 ID、IDREF 或 IDREFS 结合使用，然而不允许 XML 指令与除 hide 指令之外的任何其他指令结合使用。

cdata 指令通过将数据与 CDATA 部分包装在一起来包含数据，不对内容进行实体编码。原始数据类型必须是文本类型（如 varchar、nvarchar、text 或 ntext），此指令只适用于 hide。当使用此指令时，不能指定 AttributeName。

大多数情况下，允许在这两个组之间组合指令，但不允许在它们自身当中组合指令。如果未指定 Directive 和 AttributeName（例如 Customer!1），则隐含一个 element 指令（如 Customer!1!!element），并且列数据包含在 ElementName 中。如果指定了 XMLtext 指令，则列内容包装在与文档的其余部分集成在一起的单个标记中。在提取由 OPENXML 存储在列中的溢出（未用完的）XML 数据时，此指令很有用。

如果指定了 AttributeName，将由指定的名称替换标记名。否则，将通过把内容置于包容的起始位置（不经过实体编码）将属性追加到闭合元素属性的当前列表中。包含此指令

的列必须是文本类型（如 varchar、nvarchar、char、nchar、text 或 ntext）。此指令只适用于 hide。在提取列中存储的溢出数据时，此指令很有用。如果内容的 XML 格式不正确，则未定义该行为。

10.2.4　PATH 模式

PATH 模式提供了一种较简单的方法来混合元素和属性，其是一种用于引入附加嵌套来表示复杂属性的较简单的方法。尽管可以使用 FOR XML EXPLICIT 模式查询从行集构造这种 XML，但 PATH 模式为可能很麻烦的 EXPLICIT 模式查询提供了一种较简单的替代方法。通过 PATH 模式，以及用于编写嵌套 FOR XML 查询的功能和返回 XML 类型实例的 TYPE 指令，可以编写简单的查询。

在 PATH 模式中，列名或列别名被作为 XPath 表达式来处理。这些表达式指明了如何将值映射到 XML。每个 XPath 表达式都是一个相对的 XPath，它提供了项类型（例如属性、元素和标量值）以及将相对于行元素而生成的节点的名称和层次结构。

如果希望返回的 XML 数据是在子元素中而不是在属性中，那么可以使用 PATH 模式。将代码 10.5 中的 AUTO 改为 PATH 后，脚本和运行返回结果如代码 10.10 所示。

代码 10.10　PATH 模式

```
SELECT TOP 2 pe.EmailAddress,pp.FirstName,pp.LastName
FROM Person.Person pp
INNER JOIN Person.EmailAddress pe
ON pp.BusinessEntityID = pe.BusinessEntityID
FOR XML PATH    --指定使用 PATH 模式
```

其运行结果为：

```
<row>  <EmailAddress>ken0@adventure-works.com</EmailAddress>
  <FirstName>Ken</FirstName>
  <LastName>Sánchez</LastName>
</row>
<row>  <EmailAddress>terri0@adventure-works.com</EmailAddress>
  <FirstName>Terri</FirstName>
  <LastName>Duffy</LastName>
</row>
```

说明： 同 RAW 模式相同，PATH 模式后也可以跟参数，用来指定每一行转换后的元素名称。

10.3　SQL Server 2012 对 XML 的支持

SQL Server 2000 已经开始定义 FOR XML 子句和 OPENXML()函数来对 XML 进行操作，随着 XML 的广泛应用，SQL Server 2005 和 SQL Server 2008 也加强了对 XML 的操作

功能。本节将主要学习 FOR XML 子句和 OPENXML()函数的功能。

10.3.1　对 FOR XML 子句的增强

在 SQL Server 2005 版本中对 XML 的支持进行了增强，FOR XML 子句除了指定 XML 模式之外还可以跟其他 SQL 语句以增强对返回 XML 内容的控制。主要增强语句包括：

1．RAW模式下ELEMENTS支持

如果在 FORXML 子句中指定了可选的 ELEMENTS 选项，SELECT 子句中列出的列将映射到属性或子元素。ELEMENTS 语句在 SQL Server 2000 中就已经支持使用，但是却不能运行在 RAW 模式下，SQL Server 2005 增加了 RAW 模式下对 ELEMENTS 的支持。ELEMENTS 的使用语法如代码 10.11 所示。

代码 10.11　RAW 模式下的 ELEMENTS 支持

```
USE AdventureWorks2012;
SELECT TOP 2 at.AddressTypeID,at.Name,ca.AddressID
FROM Person.AddressType at
INNER JOIN Person.BusinessEntityAddress ca
ON ca.AddressTypeID = at.Ad BusinessEntityAddress dressTypeID
FOR XML RAW,Elements        --RAW 模式，支持 Elements
```

其运行结果为：

```
<row>
  <AddressTypeID>2</AddressTypeID>
  <Name>Home</Name>
  <AddressID>249</AddressID>
</row>
<row>
  <AddressTypeID>2</AddressTypeID>
  <Name>Home</Name>
  <AddressID>293</AddressID>
</row>
```

2．NULL支持

如果当 SQL 语句返回的结果集中有某列值为 NULL 时，那么 FOR XML 子句将不会对该 NULL 值生成对应的元素或属性，运行代码 10.12 将看到 NULL 值的 MiddleName 在第一个元素中并没有生成对应的子元素。

代码 10.12　FOR XML 对空值的处理

```
USE AdventureWorks2012;
SELECT TOP 2 FirstName,MiddleName,LastName
FROM Person. Person
FOR XML RAW ,ELEMENTS   --RAW 模式
```

其运行结果为：

```
<row>
```

```
  <FirstName>Syed</FirstName>
  <MiddleName>E</MiddleName>
  <LastName>Abbas</LastName>
</row>
<row>
  <FirstName>Catherine</FirstName>
  <MiddleName>R.</MiddleName>
  <LastName>Abel</LastName>
</row>
```

如果需要创建 NULL 值对应的元素，那么可以通过对 ELEMENTS 指令指定可选的 XSINIL 参数，将为每个 NULL 列值返回一个元素，其 xsi:nil 属性被设置为 TRUE。在代码 10.8 的 SQL 语句后面直接跟 XSINIL 语句便可以得到含有 NULL 值的 XML 结果，其脚本和结果如代码 10.13 所示。

代码 10.13　添加 XSINIL 的语句

```
USE AdventureWorks2012;
SELECT TOP 2 FirstName,MiddleName,LastName
FROM Person.Person
FOR XML RAW ,ELEMENTS XSINIL    --增加对 NULL 元素的处理
```

其运行结果为：

```
<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <FirstName>Syed</FirstName>
  <MiddleName>E</MiddleName>
  <LastName>Abbas</LastName>
</row>
<row xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <FirstName>Catherine</FirstName>
  <MiddleName>R.</MiddleName>
  <LastName>Abel</LastName>
</row>
```

3. Inline XSD Schema

前面所讲的 FOR XML 语句都是返回 XML 的结果集，若希望得到 XML 结果集的 XML Schema 信息，可以在 XML 子句后增加 XMLSCHEMA 语句。其 SQL 脚本如代码 10.14 所示。

代码 10.14　获得 XML 结果集的 XML Schema 信息

```
USE AdventureWorks2012;
SELECT TOP 2 FirstName,MiddleName,LastName
FROM Person.Person
FOR XML RAW,XMLSCHEMA('urn:example')    --获得 XML 架构信息
```

其运行结果为：

```
<xsd:schema                                targetNamespace="urn:example"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
elementFormDefault="qualified">
  <xsd:import
namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/sq
ltypes.xsd" />
  <xsd:element name="row">
    <xsd:complexType>
      <xsd:attribute name="FirstName" use="required">
        <xsd:simpleType
sqltypes:sqlTypeAlias="[AdventureWorks2012].[dbo].[Name]">
          <xsd:restriction                       base="sqltypes:nvarchar"
sqltypes:localeId="1033"          sqltypes:sqlCompareOptions="IgnoreCase
IgnoreKanaType IgnoreWidth" sqltypes:sqlSortId="52">
            <xsd:maxLength value="50" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
...
</xsd:schema>
<row xmlns="urn:example" FirstName="Syed" MiddleName="E" LastName="Abbas"
/>
<row     xmlns="urn:example"     FirstName="Catherine"     MiddleName="R."
LastName="Abel" />
```

返回结果上部分就是 XML 结果集的 XML Schema 信息，而下面就是返回的具体 XML 结果。另外，XMLSCHEMA 后跟的参数不是必需的，如果没有提供该参数，系统将在返回的结果中自动生成一个。

4．Type指明返回XML数据类型值

若希望将 FOR XML 查询的 XML 结果集作为查询中的一个列输出，从而形成一种嵌套的结构，或者需要将返回的 XML 结果存为变量以进行进一步的处理，那么可以在 FOR XML 子句后使用 TYPE 命令来表示返回结果集是 XML 数据类型。TYPE 的使用语法如代码 10.15 所示。

代码 10.15　TYPE 表示返回一个 XML 数据

```
USE AdventureWorks2012;
SELECT  at.AddressTypeID,at.Name,(
    SELECT ca.AddressID
    FROM Person. BusinessEntityAddress  ca
    WHERE ca.AddressTypeID = at.AddressTypeID
    FOR XML AUTO,TYPE--表明返回的是一个 XML 数据，而不是一个结果集
    )AS XMLCustomerAddress
FROM Person.AddressType at
```

其返回的结果如图 10.3 所示，子查询被看做一个 XML 数据作为列输出。

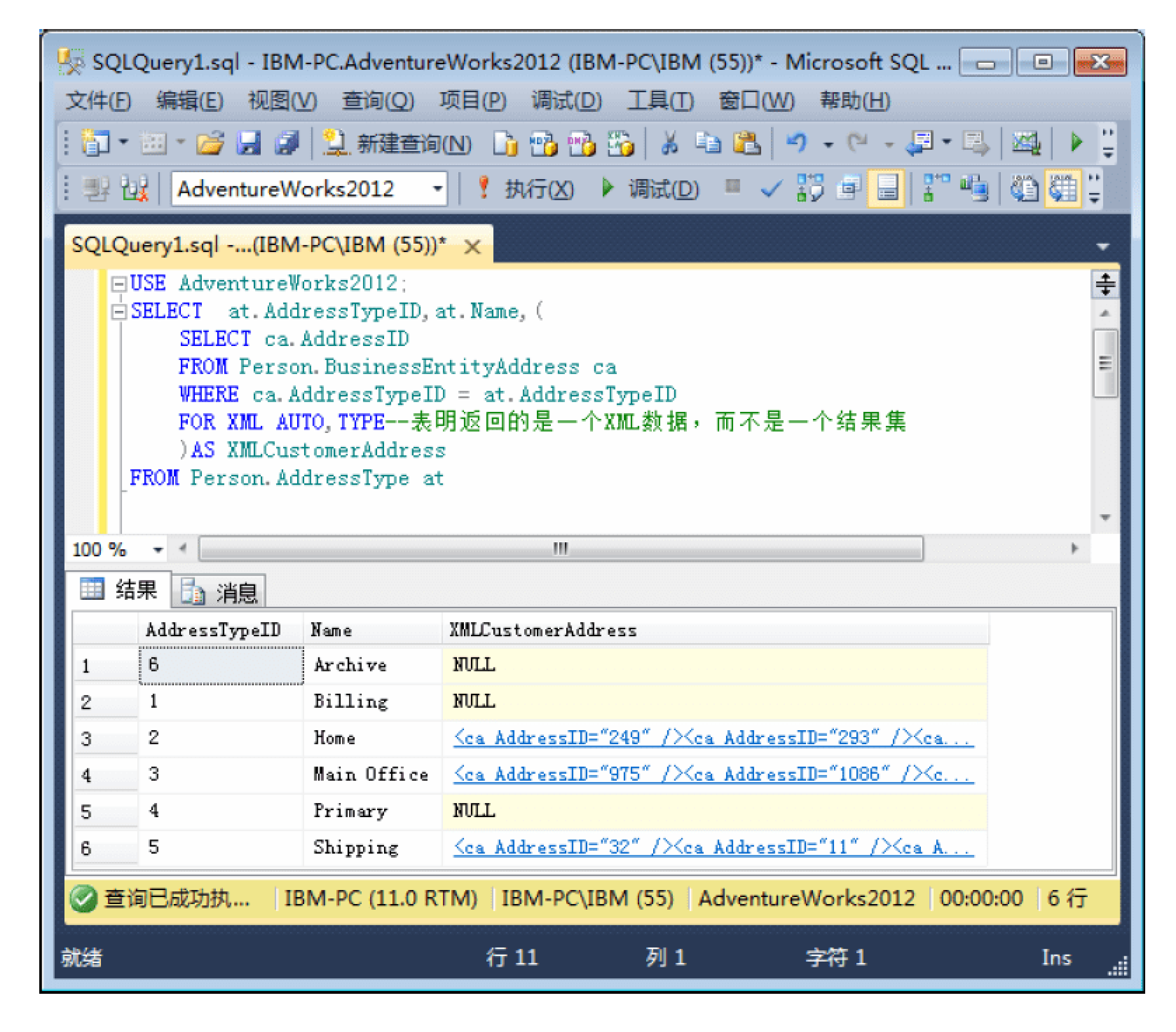


图 10.3　SSMS 下运行带 TYPE 的 FOR XML 语句

5. ROOT标识

在前面章节中笔者已经提到 XML 数据是一个树结构必须要有一个根节点，而前面所有的 FOR XML 子句返回的 XML 结果集中都没有根节点。这时，若要为 XML 结果添加根节点就要用到 ROOT 语句。ROOT 语句后可以跟参数指定根节点的元素名，若未指定则默认值为“root”。若要将用户查询返回的 XML 结果中添加 Person 作为根节点，则对应的 SQL 脚本如代码 10.16 所示。

代码 10.16　使用 ROOT 语句

```
USE AdventureWorks2012;
SELECT TOP 2 FirstName,MiddleName,LastName
FROM Person.Person
FOR XML RAW,ROOT('Person')      --指定根节点名
```

其运行的结果为：

```
<Person>
  <row FirstName="Syed" MiddleName="E" LastName="Abbas" />
  <row FirstName="Catherine" MiddleName="R." LastName="Abel" />
</Person>
```

10.3.2　OPENXML()函数

前面讲到的 FOR XML 子句都是将行结果集转换为 XML 结果集，那么如果想要将 XML 文档转换成行结果集，这时就要使用 OPENXML()函数。OPENXML()函数在 SQL Server 2000 中就已经提供，但是在 SQL Server 2005 中对该函数进行了增强。OPENXML()函数的

语法如代码 10.17 所示。

代码 10.17　OPENXML()语法

```
OPENXML( idoc int [ in] , rowpattern nvarchar [ in ] , [ flags byte [ in ] ] )
[ WITH ( SchemaDeclaration | TableName ) ]
```

第一个参数 idoc 是 XML 文档的句柄，该句柄需要通过调用 sp_XML_preparedocument 创建 XML 文档的内部表示形式来获得。参数 rowpattern 是一个 XPATH 模式，用来标识要作处理的节点。第三个参数中用 1 表示查询以属性为中心，2 表示查询以元素为中心。最后 WITH 子句标识出要返回的字段。

就以用户名 XML 结果作为 XML 文档，使用 OPENXML()函数将该 XML 文档转换为行结果集的代码如代码 10.18 所示。

代码 10.18　使用 OPENXML()函数

```
declare @mydoc XML
set @mydoc='
<Person>
  <row FirstName="Gustavo" LastName="Achong" />
  <row FirstName="Catherine" MiddleName="R." LastName="Abel" />
</Person>'
--定义 XML 文档
declare @docHandle int
Exec sp_XML_preparedocument @docHandle OUTPUT,@mydoc --获得 XML 的句柄
--获得 XML 文档的句柄
SELECT *
FROM OPENXML(@docHandle,'/Person/row',1)--1 表示以属性为中心
WITH    (FirstName    nvarchar(50),MiddleName    nvarchar(50),LastName
nvarchar(50))
```

其运行的结果为：

```
FirstName    MiddleName   LastName
----------  -----------  ---------
Gustavo      NULL          Achong
Catherine     R.           Abel
```

若将 OPENXML()函数的第三个参数换为 2 那么将返回两行 NULL 值，因为 2 表示查询以元素为中心，而 row 节点下没有其他元素。同样的道理，如果给出的 XML 文档只有元素而没有属性，那么就要使用参数 2 而不能使用 1。如果想要查询出的数据一部分在元素的属性中，一部分在元素的子元素中，则可以将该参数换成 3。查询语句及返回结果如代码 10.19 所示。

说明： 实际上第三个参数无论跟任何正整数 SQL Server 2012 都不会报错，微软官方 MSDN 中只给出了 0、1、2、8 这 4 个数字的意思。

代码 10.19　使用 OPENXML()函数查询属性和子元素

```
declare @mydoc XML
set @mydoc='
<Products>
  <Product Category="Book">
    <Name>Windows 2008</Name>
```

```
    <Vendor>Vendor1</Vendor>
  </Product>
  <Product Category="Book">
    <Name>SQL2008</Name>
    <Vendor>Vendor2</Vendor>
  </Product>
</Products>'
declare @docHandle int
Exec sp XML preparedocument @docHandle OUTPUT,@mydoc
--调用系统存储过程获得 XML 句柄
SELECT *
FROM OPENXML(@docHandle,'/Products/Product',3)
WITH (Category nvarchar(50),Name nvarchar(50),Vendor nvarchar(50))
```

其实，SQL Server 2012 内部是根据第三个参数的二进制比特位上的值来确定查询的方式。如果最后两位是 00 或 01（比如 0、1、4、5 等）就是以属性为中心进行查询；如果最后两位是 10（比如 2、6、10 等）就是以元素为中心的查询；而最后两位是 11（比如 3、7 等）表示既要查询属性也要查询元素。

10.4　XML 数据类型

XML 数据类型是 SQL Server 2005 中新增加的一种数据类型，可以将 XML 文档和片段存储在 SQL Server 数据库中。

10.4.1　XML 数据类型简介

XML 数据类型可以作为列也可以声明为变量，可以选择性地将 XML 架构集合与 XML 数据类型的列、参数或变量进行关联。集合中的架构用于验证和类型化 XML 实例。但是 XML 数据类型一般有如下限制：

- XML 数据类型实例所占据的存储空间大小不能超过 2GB。
- 不能用做 sql_variant 实例的子类型。
- 不支持转换或转换为 text 或 ntext。可改用 varchar(max)或 nvarchar(max)。
- 不能进行比较或排序。这意味着 XML 数据类型不能用在 GROUP BY 语句中。
- 不能用做除 ISNULL()、COALESCE()和 DATALENGTH()之外的任何内置标量函数的参数。
- 不能用做索引中的键列。但可以作为数据包含在聚集索引中；如果创建了非聚集索引，也可以使用 INCLUDE 关键字显式添加到该非聚集索引中。
- 除了 string 类型，没有其他数据类型能够转换成 XML。
- 分布式局部（partitioned）视图不能包含 XML 数据类型。
- XML 列不能成为主键或外键的一部分。
- XML 列不能指定为唯一的。
- COLLATE 子句不能被使用在 XML 列上。
- XML 列不能加入到规则中。

- 表中最多只能拥有 32 个 XML 列。
- 具有 XML 列的表不能有一个超过 15 列的主键。
- 具有 XML 列的表不能有一个 timestamp 数据类型作为它们主键的一部分。
- 存储在数据库中的 XML 仅支持 128 级的层次。

XML 数据类型使用关键字“XML”来表示。如代码 10.20 所示为创建含有 XML 数据类型列的表和定义 XML 数据类型的变量。

代码 10.20　XML 数据类型列和变量

```
CREATE TABLE testTb
(
    ID int PRIMARY KEY IDENTITY,
    Name nvarchar(50) NOT NULL,
    XMLCol XML NOT NULL --XML 类型的列
)
GO
DECLARE @XML XML   --XML 类型的变量
SET @XML='<student>张三</student>'
insert into testTb values(N'张三',@XML)
```

10.4.2　使用非类型化 XML

非类型化 XML 是指没有关联 XML 架构集合的 XML 类型的变量、参数或列。相反地，关联了 XML 架构集合的 XML 类型数据就叫做类型化 XML。

XML 数据类型可实现 ISO 标准的 XML 数据类型。因此，可以在非类型化的 XML 列中存储格式正确的 XML1.0 版的文档，以及具有文本节点和任意数量顶级元素的所谓的 XML 内容片段。系统将检查数据格式是否正确，但不要求将列绑定到 XML 架构，并且拒绝在扩展意义上格式不正确的数据。对于非类型化的 XML 变量和参数也是如此。在下列情况下，使用非类型化的 XML：

- 没有对应的 XML 数据的架构。
- 有架构，但不希望服务器验证数据。当应用程序将数据存储到服务器之前会执行客户端验证时；临时存储对该架构而言无效的 XML 数据时；或在服务器上使用不支持的架构组件时，需要如此。

简单地说，非类型化 XML 就是没有格式限制的 XML，允许用户向其中插入各种 XML 数据。例如创建一个学生表，该表中使用 XML 类型列来存储学生的姓名、性别和生日。由于是非类型化的 XML，所以可以向其中插入学生数据 XML 也可以向其中插入班级数据、课程数据、教师数据等。非类型化 XML 数据列和使用如代码 10.21 所示。

代码 10.21　使用非类型化 XML

```
CREATE TABLE XMLStudent
(
    StuID int IDENTITY PRIMARY KEY,
    StuInfo XML
)
GO
INSERT INTO XMLStudent --插入学生数据
```

```
VALUES(
'<Student>
    <Name>何欢</Name>
    <Sex>1</Sex>
    <Birthday>1982-08-06</Birthday>
</Student>')
INSERT INTO XMLStudent --插入了课程数据
VALUES(
'<Course>
    <Name>量子力学</Name>
    <Score>4</Score>
</Course>')
```

10.4.3　管理 XML 架构集合

XML 架构集合是一个类似于数据库表的元数据实体，可以创建、修改和删除。CREATE XML SCHEMA COLLECTION 语句中指定的架构将自动导入到新建的 XML 架构集合对象中。通过使用 ALTER XML SCHEMA COLLECTION 语句，可以将其他架构或架构组件导入到数据库中的现有集合对象中。

SQL Server 使用架构集合可以优化数据存储。查询处理引擎也使用该架构进行类型检查并优化查询和数据修改。此外，SQL Server 使用相关联的 XML 架构集合（在类型化的 XML 的情况下）来验证 XML 实例。如果 XML 实例符合架构，则数据库允许该实例存储在包含其类型信息的系统中。否则，它将拒绝该实例。为了管理数据库中的架构集合，SQL Server 提供了下列 DDL 语句：

- CREATE XML SCHEMA COLLECTION 将架构组件导入数据库。
- ALTER XML SCHEMA COLLECTION 修改现有 XML 架构集合中的架构组件。
- DROP XML SCHEMA COLLECTION 删除整个 XML 架构集合及其所有组件。

创建架构集合的语法为：

```
CREATE XML SCHEMA COLLECTION [ <relational schema>. ]sql identifier AS
Expression
```

其中 relational_schema 标识关系架构的名称。如果不指定，则假定为默认关系架构。sql_identifier 是 XML 架构集合的 SQL 标识符。Expression 为字符串常量或标量变量的架构内容，为 varchar、varbinary、nvarchar 或 XML 类型。

例如要创建学生信息 XML 的架构集合，则对应脚本如代码 10.22 所示。

代码 10.22　创建架构集合

```
CREATE XML SCHEMA COLLECTION StudentSchema AS '<?xml version="1.0"
encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormD-
efault="qualified">
    <xs:element name="Birthday" type="xs:date"/>
    <xs:element name="Name">
        <xs:simpleType>
            <xs:restriction base="xs:string">
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
    <xs:element name="Sex" type="xs:boolean"/>
```

```
    <xs:element name="Student">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Name"/>
                <xs:element ref="Sex"/>
                <xs:element ref="Birthday"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>'
```

添加后可以通过查询系统视图 sys.XML_schema_collections 获得当前数据库的架构集合信息。若要向现有 XML 架构集合中添加新架构组件，就需要使用 ALTER XML SCHEMA COLLECTION，其语法格式为：

```
ALTER XML SCHEMA COLLECTION [ relational_schema. ]sql_identifier ADD 'Schema Component'
```

其中 relational_schema 标识关系架构的名称。如果未指定，则假定为默认的关系架构。sql_identifier 是 XML 架构集合的 SQL 标识符。'Schema Component'是要插入的架构组件。

注意：如果要在集合中添加的某些组件引用同一集合中已经存在的组件时，则必须使用 <import namespace="referenced_component_namespace" />。但是，在<xsd:import>中使用当前架构命名空间是无效的，因此将自动导入与当前架构命名空间相同的目标命名空间中的组件。

若要删除整个 XML 架构集合及其所有组件，则使用 DROP XML SCHEMA COLLECTION 命令。删除架构时需要注意删除 XML 架构集合属于事务性操作，这表示如果删除事务内的 XML 架构集合然后回滚此事务，则 XML 架构集合不会被删除。当 XML 架构集合正在使用时，不能将其删除，这表示删除的集合不能存在下列任何情况：

- 与任何 XML 类型参数或列关联。
- 在任何表约束中指定。
- 被绑定到架构的函数或存储过程中引用。

10.4.4　使用类型化 XML

存储在与架构关联的列或变量中的 XML 称为类型化的 XML。在类型化 XML 中 XML 架构提供以下信息：

- 验证约束。每当向类型化的 XML 实例赋值或修改这样的实例时，SQL Server 都将验证该实例。
- 数据类型信息。架构提供有关 XML 数据类型实例中属性和元素类型的信息。与非类型化的 XML 可以提供的操作语义相比，类型信息为实例中包含的值提供了更精确的操作语义。例如，可以对十进制值执行十进制算术运算，而不能对字符串值执行十进制算术运算。因此，与非类型化的 XML 相比，可以对类型化的 XML 存储进行更大程度的压缩。

在下列情况下，使用类型化的 XML 数据类型：

- 有对应的 XML 数据的架构，并且希望服务器根据 XML 架构验证 XML 数据。

- 希望利用基于类型信息的存储和查询优化。
- 希望在编译查询过程中更好地利用类型信息。

类型化的 XML 列、参数和变量可以存储 XML 文档或内容。但是，在声明时必须使用标志指定是存储文档还是存储内容。此外，必须提供 XML 架构集合。如果每个 XML 实例都刚好有一个顶级元素，则指定 DOCUMENT；否则使用 CONTENT。查询编译器在查询编译期间的类型检查过程中，使用 DOCUMENT 标志来推断单独的顶级元素。

必须使用 CREATE XML SCHEMA COLLECTION 语句注册 XML 架构集合，然后才能创建类型化的 XML 变量、参数或列。接下来，就可以将 XML 架构集合与 XML 数据类型的变量、参数或列关联起来。同样以前面提到的学生信息 XML 为例，创建类型化 XML 列的学生表脚本如代码 10.23 所示。

代码 10.23　创建类型化 XML 列的表

```
CREATE TABLE XMLStudent2
(
    StuID int IDENTITY PRIMARY KEY,
    StuInfo XML(StudentSchema)  --使用 XML 类型的列并指定 XML 架构
)
```

该表中已经指定了 StuInfo 列的 XML 架构，所以只有符合架构的 XML 数据才能被插入。同样插入一个学生信息和一个课程信息，系统只能将学生信息插入，如代码 10.24 所示。

代码 10.24　测试类型化 XML

```
INSERT INTO XMLStudent2 --插入学生数据成功
VALUES(
'<Student>
    <Name>何欢</Name>
    <Sex>1</Sex>
    <Birthday>1982-08-06</Birthday>
</Student>')
GO
--以下插入了课程数据将报错：
--XML 验证：找不到元素'Course' 的声明。位置：/*:Course[1]
INSERT INTO XMLStudent2

VALUES(
'<Course>
    <Name>量子力学</Name>
    <Score>4</Score>
</Course>')
```

10.5　XML 类型的方法

SQL Server 除了提供 XML 类型用于存储 XML 数据外，还提供了 XML 类型的方法用于处理 XML 数据。SQL Server 提供了以下的 XML 数据类型方法：

- query()执行一个 XML 查询并且返回查询的结果。

- ❑ exists()执行一个 XML 查询，并且如果有结果的话返回值 1。
- ❑ value()计算一个查询以从 XML 中返回一个简单的值。
- ❑ modify()在 XML 文档的适当位置执行一个修改操作。
- ❑ nodes()允许把 XML 分解到一个表结构中。

本节主要讲解这几个方法的使用。

10.5.1　用 query()方法查询 XML

query()方法对 XML 数据类型的实例指定 XQuery，结果为 XML 类型。该方法返回非类型化的 XML 实例，其执行语法为：

```
query ('XQuery')
```

其中'XQuery'是一种特定类型的字符串，用于查询 XML 实例中的 XML 节点（如元素、属性）的 XQuery 表达式。关于 XQuery 的具体知识将在下面的章节中进行讲解。以前面用到的学生信息 XML 为例，若要查询一个学生信息 XML 中学生的姓名，则对应的 SQL 脚本如代码 10.25 所示。

代码 10.25　使用 query()方法查询 XML 变量

```
DECLARE @stuXML XML
SET @stuXML='<Student>
    <Name>何欢</Name>
    <Sex>1</Sex>
    <Birthday>1982-08-06</Birthday>
</Student>'
SELECT @stuXML.query('/Student/Name')  --查询 XML
```

运行后系统将查询出 XML 变量中的 Name 节点，返回“<Name>何欢</Name>”。

同样也可以将 query()方法应用于 XML 列，如查询 XMLStudent2 表中所有学生的生日，则对应的脚本如代码 10.26 所示。

代码 10.26　使用 query()方法查询 XML 列

```
SELECT StuInfo.query('/Student/Birthday') AS Birthday --XML 查询
FROM XMLStudent2
```

🔔注意：如果对 NULL XML 实例执行 XML 数据类型方法 query()、value()和 exist()，它们将返回 NULL。此外，modify()方法不返回任何值，而 nodes()方法返回行集和一个输入为 NULL 的空行集。

10.5.2　用 exists()方法判断查询是否有结果

exists()方法用于判断是否能根据 XQuery 查询出结果，如果能够得到非空结果则返回 1；如果查询得到一个空结果则返回 0；如果查询的 XML 为 NULL，则返回 NULL。exist()方法的语法格式为：

```
exist('XQuery')
```

当使用 exist()方法时，系统将计算 XQuery 查询，并且如果该查询得到任何结果，返回值都是 1。如代码 10.27 所示，定义一个 XML 变量然后使用 exists()方法，查询是否其日期时间为 2008-01-01。

代码 10.27　在 XML 变量中使用 exists()方法

```
declare @x XML
declare @f bit
set @x = '<root Somedate = "2008-01-01"/>'
set   @f   =   @x.exist('/root[(@Somedate   cast   as   xs:date?)   eq
xs:date("2008-01-01")]')
--判断其中的表达式是否为真
select @f
```

对于定义了 XML 列的表来说，exists()方法更多的时候是用于 WHERE 条件中。例如要查询学生信息表 XMLStudent 中出生日期为 1982-08-06 的所有学生，则对应的脚本如代码 10.28 所示。

代码 10.28　在 XML 列中使用 exists()方法

```
SELECT StuInfo.query('/Student/Birthday') AS Birthday
FROM XMLStudent
WHERE StuInfo.exist('/Student/Birthday[(text()[1] cast as xs:date ?) =
xs:date("1982-08-06") ]')=1
```

注意： text()方法返回一个文本节点，为 XML 标签中的文本，将其转换为日期格式后再与另外一个日期进行比较。

10.5.3　用 value()方法返回查询的原子值

当不想解释整个查询的结果而只想得到一个标量值时，value()方法是很有用的。Value()方法用于查询 XML 并且返回一个原子值，其语法格式为：

```
value(XQuery, SQLType)
```

其中 SQLType 是要返回的首选 SQL 类型（一种字符串文字）。此方法的返回类型与 SQLType 参数匹配。SQLType 不能是 XML 数据类型、公共语言运行时（CLR）用户定义类型、image、text、ntext 或 sql_variant 数据类型。SQLType 可以是用户定义数据类型 SQL。

说明： 借助于 value()方法可以从 XML 中得到单个标量值。

例如要查询学生表中所有学生的姓名，若使用 query()方法将得到“<Name>何欢</Name>”这样的形式；而使用 value()方法将不会返回 XML 标签，只返回其中的结果。具体脚本如代码 10.29 所示。

代码 10.29　使用 value()方法查询 XML 列

```
SELECT StuInfo.value('(/Student/Name)[1]','nvarchar(20)') AS StuName
```

```
--返回查询出的值
FROM XMLStudent2
```

由于性能原因，不在谓词中使用 value()方法与关系值进行比较，而改用具有 sql:column()的 exist()，如代码 10.30 所示。

代码 10.30　使用 exists()方法代替 value()方法

```
CREATE TABLE T (c1 int, c2 varchar(10), c3 XML)
GO
SELECT c1, c2, c3
FROM T
WHERE c3.value( '/root[1]/@a', 'integer') = c1  --有性能问题
GO
-- 可以写入以下语句:
SELECT c1, c2, c3
FROM T
WHERE c3.exist( '/root[@a=sql:column("c1")]') = 1  --代替方法
GO
```

10.5.4　用 modify()方法修改 XML 的内容

modify()方法用于修改 XML 文档的内容。使用此方法可修改 XML 类型变量或列的内容。此方法使用 XML DML 语句在 XML 数据中插入、更新或删除节点。XML 数据类型的 modify()方法只能在 UPDATE 语句的 SET 子句中使用。modify()方法的语法格式为：

```
modify (XML_DML)
```

其中 XML_DML 是 XML 数据操作语言（DML）中的字符串。将根据此表达式来更新 XML 文档。

注意：如果针对空值或以空值表示的结果调用 modify()方法，则会返回错误。

XML 数据修改语言（XML DML）是 XQuery 语言的扩展。根据 W3C 的定义，XQuery 语言缺少数据操作（DML）部分。XML DML 及 XQuery 语言，提供了完整的功能查询和数据修改语言，可以使用它们对 XML 数据类型进行操作。XMLDML 将下列区分大小写的关键字添加到 XQuery 中：insert、delete 和 replace value of。

在 XML DML 的操作中存在某些无法插入、删除或修改其值的属性。例如：

- 对于类型化或非类型化的 XML 而言，这样的属性有 XMLns、XMLns:*和 XML:base。
- 仅对于类型化的 XML 而言，这样的属性有 xsi:nil 和 xsi:type。

另外还存在以下限制：

- 对于类型化或非类型化的 XML，插入 XML:base 属性将失败。
- 对于类型化的 XML，删除和修改 xsi:nil 属性将会失败；对于非类型化的 XML，则可以删除此属性或修改此属性的值。
- 对于类型化的 XML，修改 xs:type 属性值将会失败；对于非类型化的 XML，则可以修改此属性值。

1．插入XML DML

将 Expression1 标识的一个或多个节点，作为 Expression2 标识节点的子节点或同级节点插入需要使用 insert 命令，其语法格式如代码 10.31 所示。

代码 10.31　XML DML 中的 insert 语法

```
insert
    Expression1 (
           {as first | as last} into | after | before
                        Expression2
          )
```

其中每个参数的含义说明如下。

- Expression1：标识要插入的一个或多个节点。其可以是一个常量 XML 实例、对应的相同 XML 架构集合的 XML 数据类型实例，也可以是使用单独的 sql:column()/sql:variable()函数的非类型化 XML 数据类型实例，或者是一个 XQuery 表达式。该表达式可以得出节点、文本节点或一组有序的节点，但它无法解得根（/）节点。如果该表达式得出一个值或一组值，则这些值作为单个文本节点插入，各值之间以空格分隔开。如果将多个节点指定为常量，则这些节点用括号括住，并以逗号分隔开，但无法插入异构序列（如一组元素、属性或值）。如果 Expression1 解得一个空序列，则不会发生插入操作，并且不会返回任何错误。
- into：表示 Expression1 标识的节点作为 Expression2 标识节点的直接后代（子节点）插入。如果 Expression2 中的节点已有一个或多个子节点，则必须使用 as first 或 as last 指定所需的新节点添加位置，例如分别在子列表的开头或末尾。插入属性时忽略 as first 和 as last 关键字。
- after：表示 Expression1 标识的节点作为 Expression2 标识节点的同级节点直接插入在其后面。after 关键字不能用于插入属性。例如，它不能用于插入属性构造函数或从 XQuery 中返回属性。
- before：表示 Expression1 标识的节点作为 Expression2 标识节点的同级节点直接插入在其前面。before 关键字不能用于插入属性。例如，它不能用于插入属性构造函数或从 XQuery 中返回属性。
- Expression2：标识节点。Expression1 标识的节点是相对于 Expression2 标识的节点插入的，其可以是 XQuery 表达式，返回当前被引用的文档中现有节点的引用。如果返回多个节点，则插入失败；如果 Expression2 返回一个空序列，则不会发生插入操作，并且不会返回任何错误；如果 Expression2 在静态时不是单独的，则将返回静态错误。Expression2 不能是处理指令、注释或属性。请注意，Expression2 必须是文档中现有节点的引用，并且不能是构造的节点。

同样以前面使用的学生 XML 数据为例，若需要将学号插入到 Student 节点下作为其子节点，则对应的 SQL 语句如代码 10.32 所示。

代码 10.32　XML 插入子节点

```
DECLARE @stuXML XML --定义 XML 变量
SET @stuXML='<Student>
    <Name>何欢</Name>
```

```
    <Sex>1</Sex>
    <Birthday>1982-08-06</Birthday>
</Student>'
--接下来修改 XML 变量中的内容
SET @stuXML.modify('
insert <StudentID>2303002027</StudentID>
as first
into (/Student)[1]
')
SELECT @stuXML
```

其返回结果为：

```
<Student>
  <StudentID>2303002027</StudentID>
  <Name>何欢</Name>
  <Sex>1</Sex>
  <Birthday>1982-08-06</Birthday>
</Student>
```

2．替换XML DML

若要在文档中更新节点的值，则需要使用 replace value of 语句，其语法格式如代码 10.33 所示。

代码 10.33　replace value of 的语法格式

```
replace value of
     Expression1
with
     Expression2
```

其中 Expression1 标识其值要更新的节点。它必须仅标识一个单个节点。如果 XML 已类型化，则节点的类型必须是简单类型。如果选择了多个节点，则会出现错误。如果 Expression1 返回空序列，则不会发生值替换，也不会返回错误。Expression1 必须返回具有简单类型内容（列表或原子类型）的单个元素、文本节点或属性节点。Expression1 不可能是联合类型、复杂类型、处理指令、文档节点或注释节点。如果它是，则会返回错误。

Expression2 标识节点的新值。这可能是返回简单类型节点的表达式，因为将隐式使用 data()。如果该值是值列表，则 update 语句将使用此列表替换旧值。在修改类型化的 XML 实例中，Expression2 必须是 Expression1 的相同类型或子类型；否则，将返回错误。在修改非类型化的 XML 实例中，Expression2 必须是可以进行原子化的表达式；否则，将返回错误。例如要修改学生信息 XML 数据中的性别，则可以使用 replace value of 语句实现，具体如代码 10.34 所示。

代码 10.34　使用 replace value of 更新性别

```
DECLARE @stuXML XML
SET @stuXML='<Student>
    <Name>何欢</Name>
    <Sex>1</Sex>
    <Birthday>1982-08-06</Birthday>
</Student>'
SET @stuXML.modify('
replace value of (/Student/Sex/text())[1]
```

```
with 0
')--替代性别中的值为 0
SELECT @stuXML
```

修改后系统返回的 XML 为：

```
<Student>
  <Name>何欢</Name>
  <Sex>0</Sex>
  <Birthday>1982-08-06</Birthday>
</Student>
```

3. 删除XML DML

删除 XML 实例的节点使用 delete 语句，其语法格式为：

```
delete Expression
```

其中 Expression 识别要删除节点的 XQuery 表达式。删除该表达式选择的所有节点，以及所选节点中的所有节点或值。如 insert(XML DML)中所介绍的，在文档中必须保持对现有节点的引用，不能是构造的节点，表达式不能是根（/）节点。如果表达式返回空序列，则不删除，不返回错误。例如要从学生信息 XML 中删除生日节点，则对应的脚本如代码 10.35 所示。

代码 10.35　使用 delete 删除 XML 中的节点

```
DECLARE @stuXML XML
SET @stuXML='<Student>
    <Name>何欢</Name>
    <Sex>1</Sex>
    <Birthday>1982-08-06</Birthday>
</Student>'
SET @stuXML.modify('
delete /Student/Birthday
')  --删除指定 XML 节点
SELECT @stuXML
```

系统返回的 XML 为：

```
<Student>
  <Name>何欢</Name>
  <Sex>1</Sex>
</Student>
```

10.5.5　用 nodes()方法实现 XML 数据到关系数据的转变

nodes()方法用于把一组由一个查询返回的结点，转换成一个类似于结果集的表中的一组记录行。如果要将 XML 数据类型实例拆分为关系数据，则 nodes()方法非常有用，它允许标识将映射到新行的节点。

每个 XML 数据类型实例都有隐式提供的上下文节点。对于在列或变量中存储的 XML 实例来说，它是文档节点。文档节点是位于每个 XML 数据类型实例顶部的隐式节点。

nodes()方法的结果是一个包含原始 XML 实例的逻辑副本的行集。在这些逻辑副本中，

每个行示例的上下文节点都被设置成由查询表达式标识的节点之一。这样，后续的查询可以浏览与这些上下文节点相关的节点。该方法的语法如下：

```
nodes (XQuery) as Table(Column)
```

其中 XQuery 是字符串文字，即一个 XQuery 表达式。如果查询表达式构造节点，这些已构造的节点将在结果行集中显示。如果查询表达式生成一个空序列，则行集将为空。如果查询表达式静态生成一个包含原子值而不是节点的序列，将产生静态错误。Table(Column) 指定结果行集的表名称和列名称。例如定义了多个学生信息在 XML 中，现在需要将所有学生的名字取出作为一个表返回，则对应的脚本如代码 10.36 所示。

代码 10.36　使用 nodes()方法获得 XML 中的节点

```
DECLARE @stuXML XML
SET @stuXML='<Student>
    <Name>何欢</Name>
    <Sex>1</Sex>
    <Birthday>1982-08-06</Birthday>
</Student>
<Student>
    <Name>晏婉</Name>
    <Sex>0</Sex>
    <Birthday>1974-07-06</Birthday>
</Student>
'
SELECT T.c.query('.') as StuName--获得节点
FROM @stuXML.nodes('/Student/Name/text()') AS T(c) --获得指定路径的节点
```

系统将返回所有学生姓名的列表。

使用 nodes()方法时需要注意返回的行集已对类型信息进行了维护。可以将 XML 数据类型方法（例如 query()、value()、exist()和 nodes()）应用于 nodes()方法的结果。但是不能将 modify()方法用于修改 XML 实例。另外，行集中的上下文节点无法具体化，即无法在 SELECT 语句中使用此节点。但是可以在 IS NULL 和 COUNT(*)中使用它。

使用 nodes()方法的情况与使用 OPENXML(Transact-SQL)的情况相同。这就提供了 XML 的行集视图。但是，当在包含 XML 文档的若干行的表中使用 nodes()方法时，无须使用游标。由 nodes()方法返回的行集是未命名的行集，因此，必须通过使用别名来显式命名。

nodes()函数不能直接应用于用户定义函数的结果。若要将 nodes()函数用于标量用户定义函数的结果，可以将该用户定义函数的结果分配给一个变量，也可以使用派生表为该用户定义函数的返回值分配一个列别名，然后使用 CROSS APPLY 从该别名中选择。

10.6　XML 索引

对于 XML 数据类型列可以创建 XML 索引。它们对列中 XML 实例的所有标记、值和路径进行索引，从而提高查询性能。本节将介绍 XML 索引的基础知识和对 XML 索引的创建、修改和删除操作。

10.6.1　XML 索引简介

XML 实例作为二进制大型对象（BLOB）存储在 XML 类型列中。这些 XML 实例可以很大，并且存储的 XML 数据类型实例的二进制表示形式最大可以为 2GB。如果没有索引，则运行时将拆分这些二进制大型对象以计算查询，此拆分可能非常耗时。

在对表中的 XML 列进行 query()、exists()等 XML 类型的方法查询时，每行中的 XML 二进制大型对象将在运行时拆分，然后，计算 XML 类型方法中的 XQuery 表达式。此运行时拆分有可能开销较大，这取决于存储在列中的实例的大小和数目。

如果在应用程序环境中经常查询 XML 二进制大型对象，则对 XML 类型列创建索引很有用。但是，在数据修改过程中维护索引会带来开销。在下列情况下，应该创建 XML 索引：

- 对 XML 列进行查询在工作中很常见。必须考虑数据修改过程中的 XML 索引维护开销。
- XML 值相对较大，而检索的部分相对较小。生成索引避免了在运行时分析所有数据，并能实现高效的查询处理，从而使索引查找受益。

XML 索引分为两个类别：主 XML 索引和辅助 XML 索引。主 XML 索引是 XML 类型列的第一个索引，没有主 XML 索引就不能创建辅助 XML 索引。使用主 XML 索引时，支持 PATH、VALUE 和 PROPERTY 类型的辅助索引。根据查询类型的不同，这些辅助索引可能有助于改善查询性能。

注意：除非为使用 XML 数据类型正确设置了数据库选项，否则无法创建或修改 XML 索引。

1. 主XML索引

主 XML 索引对 XML 列中 XML 实例内的所有标记、值和路径进行索引。在创建主 XML 索引之前，需要确保相应 XML 列所在的表必须对该表的主键创建了聚集索引。因为 SQL Server 将使用此主键将主 XML 索引中的行与包含此 XML 列的表中的行进行关联。

对于列中的每个 XML 二进制大型对象，索引将创建数个数据行。主 XML 索引是 XML 数据类型列中 XML BLOB 的已拆分和持久的表示形式。主 XML 索引中的行数大约等于 XML 二进制大型对象中的节点数。当查询检索完整的 XML 实例时，SQL Server 会提供此 XML 列中的实例。XML 实例中的查询使用主 XML 索引，并可以通过使用索引本身返回标量值或 XML 子树。主 XML 索引中存储以下节点信息：

- 标记名（如元素名称或属性名称）。
- 节点值。
- 节点类型（如元素节点、属性节点或文本节点）。
- 文档顺序信息（由内部节点标识符表示）。
- 从每个节点到 XML 树的根的路径。搜索此列可获得查询中的路径表达式。
- 基表的主键。基表的主键将复制到主 XML 索引中，用于向后和基表进行连接，并且基表主键中的最大列数限制为 15。

主 XML 索引上的节点信息用于计算和构造指定查询的 XML 结果。出于优化的需要，标记名和节点类型信息编码为整数值，且 Path 列使用同样的编码。另外，路径以相反的顺序存储，以便在仅知道路径后缀的情况下能够匹配路径。

当涉及 XML 数据类型方法的查询时，查询处理器将使用主 XML 索引，并返回主索引自身中的标量值或 XML 子树。

2. 辅助XML索引

由于主 XML 索引存储的数据内容有限，为了增强搜索性能，可以创建辅助 XML 索引。必须有了主 XML 索引才能创建辅助索引，辅助索引的类型如下：

- ❑ PATH 辅助 XML 索引。
- ❑ VALUE 辅助 XML 索引。
- ❑ PROPERTY 辅助 XML 索引。

以下为创建一个或多个辅助索引的一些准则：

- ❑ 如果工作负荷对 XML 列大量使用路径表达式，则 PATH 辅助 XML 索引可能会提高工作负荷的处理速度。例如在 WHERE 子句中对 XML 列使用 exist()方法。
- ❑ 如果工作负荷通过使用路径表达式从单个 XML 实例中检索多个值，则在 PROPERTY 索引中聚集各个 XML 实例中的路径可能会很有用。这种情况通常出现在属性包方案中，此时提取对象的属性并且已知其主键值。
- ❑ 如果工作负荷涉及查询 XML 实例中的值，但不知道包含哪些值的元素名称或属性名称，则可以创建 VALUE 索引。这通常出现在 descendant 轴查找中，例如//author[last-name="Howard"]，其中<author>元素可以出现在层次结构的任何级别上。这种情况也出现在通配符查询中，例如/book[@*="novel"]，其中查询将查找具有某个值为“novel”属性的<book>元素。

如果对 XML 类型列指定路径表达式进行查询时，则 PATH 辅助索引可以提高搜索的速度。当查询在 WHERE 子句中指定 exist()方法时主索引非常有用。在此基础上如果添加 PATH 辅助索引，则还可以改善此类查询的搜索性能。

VALUE 索引的键列是主 XML 索引的节点值和路径。如果查询是基于值的查询，例如/Root/ProductDescription/@*[.="Bike"]或//ProductDescription[@Name="Bike"]，这里并没有完全指定路径或路径包含有通配符，工作负荷涉及查询 XML 实例中的值，但不知道包含这些值的元素名称或属性名称，则生成基于主 XML 索引中的节点值所创建的辅助 XML 索引，可以更快地获得结果。

PROPERTY 索引是对主 XML 索引的列（PK、Path 和节点值）创建的，其中 PK 是基表的主键。从单个 XML 实例检索一个或多个值的查询适用 PROPERTY 索引。当使用 XML 类型的 value()方法检索对象属性并且知道对象的主键值时，会发生这种情况。

10.6.2　创建 XML 索引

创建 XML 索引与创建一般数据的索引相似，使用 CREATE INDEX 语句来创建。创建 XML 索引时注意下列事项：

- ❑ 若要创建主 XML 索引，含有被索引的 XML 列的表必须具有主键的聚集索引。这

样确保在对基表进行了分区的情况下，可以使用相同的分区方案和分区函数对主 XML 索引进行分区。

- 如果存在 XML 索引，则不能修改基表的聚集主键。在修改主键之前，必须删除表的所有 XML 索引。
- 可以对单个 XML 类型列创建主 XML 索引，但无法将 XML 类型列作为键列来创建任何其他类型的索引。但是，可以在非 XML 索引中包含 XML 类型列。
- 表中的每个 XML 类型列都可以有自己的主 XML 索引。但是，一个 XML 类型列只允许有一个主 XML 索引。
- XML 索引和非 XML 索引存在于相同的命名空间中。因此，同一表的 XML 索引和非 XML 索引不能具有相同的名称。
- 对于 XML 索引，IGNORE_DUP_KEY 选项和 ONLINE 选项始终设置为 OFF。可以将这些选项的值指定为 OFF。
- 将用户表的文件组和分区信息应用于 XML 索引。用户无法单独为 XML 索引指定这些信息。
- DROP_EXISTING 索引选项可以删除主 XML 索引并创建一个新的主 XML 索引，或者删除辅助 XML 索引并创建一个新的辅助 XML 索引。但是，此选项不能通过删除辅助 XML 索引来创建新的主 XML 索引，反之亦然。
- 主 XML 索引名称与视图名称有相同的限制，而且必须唯一。

与一般索引不同的是，不能对视图中的 XML 类型列、XML 类型列的表值变量或 XML 类型变量创建 XML 索引。

若要更新表，使用 ALTER COLUMN 选项将 XML 类型列从非类型化的 XML 更改为类型化的 XML，或者从类型化的 XML 更改为非类型化的 XML，则列不应存在 XML 索引。如果确实存在，则在尝试更改列类型之前必须删除该索引。

创建 XML 索引时必须将选项 ARITHABORT 设置为 ON。若要使用 XML 数据类型方法，查询、删除、更新 XML 列中的值或向 XML 列中插入值，则必须对连接设置相同的选项。如果没有设置，则 XML 数据类型方法将会失败。

注意：有关 XML 索引的信息可以在目录视图中找到。但是与一般索引不同，XML 索引不支持 sp_helpindex。

例如有班级表，该表中有班级 ID、班级名和以 XML 存储的班级中的所有学生的信息，在该表中创建 XML 主索引的脚本如代码 10.37 所示。

代码 10.37　创建 XML 主索引

```
CREATE TABLE  Class
(
ClassID int IDENTITY PRIMARY KEY,
ClassName nvarchar(20) NOT NULL,
Students XML NOT NULL
)
GO
CREATE PRIMARY XML INDEX idx_Students --创建 XML 主索引
ON Class(Students)
```

使用 CREATE INDEX 语句可创建辅助 XML 索引，并且可指定所需的辅助 XML 索引的类型。

创建辅助 XML 索引时应注意下列事项：

- 除了 IGNORE_DUP_KEY 和 ONLINE 之外，允许对辅助 XML 索引使用所有适用于非聚集索引的索引选项。对于辅助 XML 索引，这两个选项必须始终设置为 OFF。
- 辅助索引的分区方式类似于主 XML 索引。
- DROP_EXISTING 可以删除用户表的辅助索引并为用户表创建其他辅助索引。

同样地，在 Class 表中创建 Students 列的辅助索引如代码 10.38 所示。

代码 10.38　创建 XML 辅助索引

```
CREATE XML INDEX idx_Students_PATH ON Class(Students) --创建 XML 辅助索引
USING XML INDEX idx_Students
FOR PATH
GO
CREATE XML INDEX idx_Students_VALUE ON Class(Students) --创建 XML 辅助索引
USING XML INDEX idx_Students
FOR VALUE
GO
CREATE XML INDEX idx_Students_PROPERTY ON Class(Students)--创建 XML 辅助索引
USING XML INDEX idx_Students
FOR PROPERTY
GO
```

10.6.3　修改与删除 XML 索引

建立索引后若需要修改索引则使用 ALTER INDEX 语句。ALTER INDEX 语句可用于修改现有的 XML 和非 XML 索引。但是，并非所有的 ALTER INDEX 选项都适用于 XML 索引。修改 XML 索引时以下选项不可用：

- 对于 XML 索引，重新生成和设置选项 IGNORE_DUP_KEY 无效。对于辅助 XML 索引，重新生成选项 ONLINE 必须设置为 OFF。在 ALTER INDEX 语句中，不允许 DROP_EXISTING 选项。
- 对用户表中的主键约束进行的修改，不会自动传播到 XML 索引中。用户必须首先删除 XML 索引，然后再重新创建它们。
- 如果指定了 ALTER INDEX ALL，则它将应用于非 XML 索引和 XML 索引。指定的索引选项可能对两种索引无效。在这种情况下，整个语句将失败。

例如对于前面创建的 XML 主索引，现需要修改该索引，通过将选项 ALLOW_ROW_LOCKS 设置为 OFF 来修改该索引。当 ALLOW_ROW_LOCKS 为 OFF 时，不会锁定行，并且可以使用页级锁和表级锁获得对指定索引的访问权限。修改所用的脚本如代码 10.39 所示。

代码 10.39　修改 XML 索引

```
ALTER INDEX idx_Students ON Class
SET (ALLOW_ROW_LOCKS = OFF)
```

默认情况下 XML 索引是启用的，可以通过 ALTER INDEX 语句禁用 XML 索引和重

新启用 XML 索引。禁用 XML 索引使用 DISABLE 选项，重新启用 XML 索引使用 REBUILD 选项。例如要禁用 XML 主索引 idx_Students，然后再重新启用该索引，则对应的脚本如代码 10.40 所示。

代码 10.40　禁用和重新启用 XML 索引

```
ALTER INDEX idx_Students
ON  Class
DISABLE --禁用索引
GO
ALTER INDEX idx Students
ON  Class
REBUILD --重建索引
```

注意：禁用了 XML 主索引将导致所有 XML 辅助索引被禁用。而重新启用 XML 主索引将不会自动启动 XML 辅助索引，需要单独使用 SQL 命令来启用 XML 辅助索引。

DROP INDEX 语句可用于删除现有的主（或辅助）XML 索引和非 XML 索引。但是，任何 DROP INDEX 选项都不会应用于 XML 索引。如果删除主 XML 索引，则会删除任何现有的辅助索引。

例如要删除 Class 表中的 XML 主索引，则对应的脚本为：

```
DROP INDEX idx_Students ON Class
```

10.7　使用 XQuery

XQuery 是一种可以查询结构化或半结构化 XML 数据的语言。由于数据库引擎中提供 XML 数据类型支持，因此可以将文档存储在数据库中，然后使用 XQuery 进行查询。本节将主要讲解 XQuery 的基础知识和在 SQL Server 中如何使用 XQuery。

10.7.1　XQuery 基础

XQuery 相对于 XML 的关系，等同于 SQL 相对于数据库表的关系。XQuery 被设计用来查询 XML 数据，不仅仅限于 XML 文件，还包括任何可以 XML 形态呈现的数据，包括数据库。关于 XQuery 的定义，一般解释为：

- XQuery 是用于 XML 数据查询的语言。
- XQuery 对 XML 的作用类似于 SQL 对数据库的作用。
- XQuery 被构建在 XPath 表达式之上。
- XQuery 被所有主要的数据库引擎支持（IBM DB2、Oracle、Microsoft SQL Server 等）。
- XQuery 是 W3C 标准。

XQuery 可被用来提取信息，以便在网络服务中使用、生成摘要报告、把 XML 数据转换为 XHTML 和为获得相关信息而搜索网络文档。

XQuery1.0 在 2007 年 1 月 23 日被确立为 W3C 推荐标准。XQuery 与多种 W3C 标准相兼容，比如 XML、Namespaces、XSLT、XPath 及 XML Schema。

XQuery 基于现有的 XPath 查询语言，并支持更好的迭代、更好的排序结果及构造必需的 XML 的功能。XQuery 在 XQuery 数据模型上运行，此模型是 XML 文档及可能为类型化，也可能为非类型化的 XQuery 结果的抽象概念类型信息。基于 W3C XML 架构语言所提供的类型如果没有可用的类型化信息，XQuery 将按照非类型化处理数据，这与 XPath1.0 版处理 XML 的方式相似。

1．序列和QName

在 XQuery 中，表达式的结果是由一系列 XML 节点和 XSD 原子类型的实例组成的序列。序列中的单个项称为一项。

序列中的项可以是下列之一：

- 一个节点，如元素、属性、文本、处理指令、注释或文档。
- 一个原子值，如 XSD 简单类型的实例。

例如代码 10.41 所示，查询中构造了一个由两个元素节点项组成的序列。

代码 10.41　在查询中构造序列

```
SELECT Instructions.query('
    <step1> Step 1 </step1>,
    <step2> Step 2 </step2>
') AS Result    --构造序列
FROM Production.ProductModel
WHERE ProductModelID=7
```

其返回结果为：

```
<step1> Step 1 </step1>
<step2> Step 2 </step2>
```

在这个查询中，<step1>构造结尾和<step2>之间的逗号是序列构造符，是必需的。结果中添加的空格只为说明使用，并且包含在本文档的所有示例结果中。

在使用序列时需要注意：

- 如果某个查询导致了包含其他序列的一个序列，则被包含的序列将简化为容器序列。例如，在数据模型中将序列((1,2,(3,4,5)),6)简化成(1,2,3,4,5,6)。
- 空序列是不包含任何项的序列，被表示为“()”。
- 只包含一项的序列可视为原子值，比如(1)=1。

前面说到，序列分为节点序列和原子序列。在这样的实现中，序列必须是同类的。也就是说，或者具有一个原子值序列，或者具有一个节点序列。例如代码 10.4.2 所示为有效的序列。

代码 10.42　有效的序列

```
DECLARE @x XML
SET @x = ''
SELECT @x.query('1')
```

```
SELECT @x.query('"abc", "xyz"')
SELECT @x.query('data(1)')
SELECT @x.query('<x> {1+2} </x>')
```

如果既有原子值又有节点则不能称为序列，在查询中将会报错。例如下面这个查询就是一个错误的查询。

```
DECLARE @x XML
SELECT @x.query('1,<a>ss</a>') --错误的序列
```

XQuery 中的每个标识符都是一个 QName。QName 由一个命名空间前缀和一个本地名称组成。在这样的实现中，XQuery 中的变量名是 QName，它们不能带有前缀。例如在代码 10.43 中，在表达式(/Root/a)中，Root 和 a 是 QName。

代码 10.43　Root 和 a 就是 QName

```
DECLARE @x XML
SET @x = '<Root><a>ss</a></Root>'
SELECT @x.query('/Root/a') --使用 Qname
```

再如在代码 10.44 的查询中，查询将在第一个工作中心位置迭代所有的<step>元素。

代码 10.44　XQuery 查询

```
SELECT Instructions.query('
   declare namespace AWMI="http://schemas.microsoft.com/sqlserver/
   2004/07/adventure-works/ProductModelManuInstructions";
for $Step in /AWMI:root/AWMI:Location[1]/AWMI:step
      return
           string($Step)
') AS Result --Xquery 查询的结果
FROM Production.ProductModel
WHERE ProductModelID=7
```

在代码 10.44 中，AWMIroot、AWMI:Location、AWMI:step 和$Step 都是 QNames。AWMI 是一个前缀，root、Location 和 Step 都是本地名称。$step 变量是一个 QName 并且没有前缀。SQL Server 中对 XQuery 预定义的命名空间不需要再声明就可以使用，而如果是自定义的命名空间则需要声明。如表 10.1 列出了 SQL Server 中预定义的命名空间。

表 10.1　已经预定义的命名空间

前　　缀	URI
Xs	http://www.w3.org/2001/XMLSchema
Xsi	http://www.w3.org/2001/XMLSchema-instance
Xdt	http://www.w3.org/2004/07/xpath-datatypes
Fn	http://www.w3.org/2004/07/xpath-functions
（无前缀）	urn:schemas-microsoft-com:XML-sql
Sqltypes	http://schemas.microsoft.com/sqlserver/2004/sqltypes
XML	http://www.w3.org/XML/1998/namespace
（无前缀）	http://schemas.microsoft.com/sqlserver/2004/SOAP

2．表达式上下文

表达式的上下文是用于分析和计算表达式的信息。计算 XQuery 分为以下两个阶段：

- 静态上下文，这是查询编译阶段。根据可用信息，有时会在对查询进行静态分析的过程中产生错误。
- 动态上下文，这是查询执行阶段。即使查询没有静态错误（如语法有误造成在查询编译过程中产生的错误），查询也可能在执行过程中返回错误。

静态上下文初始化指的是将有关对表达式进行静态分析的所有信息放在一起的过程。在静态上下文初始化中，要完成下列内容：

- 将“边界空格”策略设置为 strip。这样在查询中，anyelement 和 attribute 构造函数不保留边界空格。例如代码 10.45 所示，“<a>{"aa"}”将被显示为“<a>"aa"”，同样地，“ {"bb"}”也被显示为“bb”。如代码 10.45 所示，表达式上下文边界的空格被过滤。

代码 10.45　表达式上下文边界的空格被过滤

```
declare @x xml
set @x=''
select @x.query('<a>  {"aa"}  </a>,
   <b> {"bb"   }    </b>') --空格将会被过滤
```

系统返回结果：

```
<a>aa</a>
<b>bb</b>
```

- 为下列内容初始化前缀和命名空间绑定：
  - 一组预定义的命名空间。
  - 使用 WITH XMLNAMESPACES 定义的所有命名空间。
  - 查询 Prolog 中定义的所有命名空间。
- 如果查询类型化的 XML 列或变量，将与该列或变量关联的 XML 架构集合的组件导入到静态上下文中。
- 对于导入架构中的每个原子类型，也可在静态上下文中使用转换函数。

以上是初始化静态上下文，初始化后将分析（编译）查询表达式。静态分析涉及查询分析、解析在表达式中指定的函数、类型名称和对查询执行静态类型化。这样可确保查询类型安全。

下面是与静态上下文有关的限制：

- 不支持 XPath 兼容模式。
- 对于 XML 构造，仅支持 strip 构造模式，这是默认设置。因此，已构造元素节点的类型为 xdt:untyped 类型，并且属性为 xdt:untypedAtomic 类型。
- 仅支持 ordered 排序模式。
- 仅支持 strip XML 空格策略。
- 不支持基准 URI 功能。
- 不支持 fn:doc()。

- 不支持 fn:collection()。
- 不提供 XQuery 静态标记。
- 使用与 XML 数据类型关联的排序规则。将此排序规则始终设置为 Unicode 码位排序规则。

动态上下文是查询的执行阶段，指的是在执行表达式时必须可用的信息。除了静态上下文以外，在动态上下文初始化中，还要完成初始化上下文项、上下文位置和上下文大小等表达式的主要项，如下所示。

- XML 数据类型将上下文项（正在处理的节点）设置为文档节点。
- 上下文位置（即上下文项相对于正在处理的节点的位置）首先设置为 1。
- 上下文大小（正在处理的序列中的项数）首先设置为 1，因为始终有一个文档节点。

注意：*所有这些值都可由 nodes()方法覆盖。*

下面是与动态上下文有关的限制：

- 不支持“当前日期和时间”上下文函数 fn:current-date、fn:current-time 和 fn:current-dateTime。
- Implicit timezone 固定为 UTC+0，不能更改。
- 不支持 fn:doc()函数。所有查询都针对 XML 类型列或变量执行。
- 不支持 fn:collection()函数。

3．原子化

原子化是提取项的类型化值的进程。在有些环境下，原子化进程是隐式进行的。算术运算符和比较运算符等 XQuery 运算符依赖于此进程。例如，将算术运算符直接应用于节点时，通过隐式调用数据函数，首先检索节点的类型化值，这将把原子值作为操作数传递给算术运算符。如代码 10.46 所示，查询将返回 LaborHours 属性的总数。在本例中，data()函数被隐式应用于属性节点。

代码 10.46　原子化

```
declare @x XML
set @x='<ROOT>
<Location LID="1" SetupTime="1.1" LaborHours="3.3" />
<Location LID="2" SetupTime="1.0" LaborHours="5" />
<Location LID="3" SetupTime="2.1" LaborHours="4" />
</ROOT>'
SELECT @x.query('sum(/ROOT/Location/@LaborHours)') --获得指定对象的总计
--相当于：
SELECT @x.query('sum(data(ROOT/Location/@LaborHours))')
```

隐式原子化在使用算术运算符时比较常用。例如+运算符需要原子值，将隐式应用 data()函数来检索 LaborHours 属性的原子值，对 ProductModel 表中的类型为 XML 的 Instructions 列指定查询。如代码 10.47 所示，+运算符就对 LaborHours 属性进行了隐式原子化。

代码 10.47　使用+运算符的原子化

```
SELECT Instructions.query('
   declare namespace AWMI="http://schemas.microsoft.com/sqlserver/
```

```
   2004/07/adventure-works
/ProductModelManuInstructions";
for $WC in /AWMI:root/AWMI:Location[1]
        return
            <WC OriginalLaborHours = "{ $WC/@LaborHours }"
                UpdatedLaborHoursV1 = "{ $WC/@LaborHours + 1 }"
                UpdatedLaborHoursV2 = "{ data($WC/@LaborHours) + 1 }" >
            </WC>') as Result--在 XML 查询中使用+运算符进行原子化
FROM Production.ProductModel
where ProductModelID=7
```

返回结果为：

```
<WC OriginalLaborHours="2.5" UpdatedLaborHoursV1="3.5" UpdatedLabor-
HoursV2="3.5" />
```

在该查询中，请注意下列情况：

- 构造 OrignialLaborHours 属性时，原子化隐式应用于$WC/@LaborHours 返回的单独序列。LaborHours 属性的类型化值被分配给 OrignialLaborHours。
- 在构造 UpdatedLaborHoursV1 属性时，算术运算符需要原子值。因此，data()函数隐式应用于$WC/@LaborHours 返回的 LaborHours 属性。然后，原子值 1 添加到该属性。属性 UpdatedLaborHoursV2 的构造显示了 data()函数的显式应用。

注意：原子化容易导致简单类型、空集、静态类型的实例错误。原子化还会发生在传递到函数的比较表达式参数、函数返回的值、cast()函数表达式和子句按顺序传递的排序表达式中。

4．有效的布尔值

XQuery 中对布尔值的定义与 JavaScript 中的定义类似，如果操作数是空序列或布尔 false，则值为 false。否则，值为 true。如代码 10.48 所示，示例中由于表达式/a[1]返回了空序列，因此有效的布尔值为 false，下一个表达式/b[1]返回了非空序列，因此有效的布尔值为 true。

代码 10.48　布尔值

```
DECLARE @x XML
SET @x = '<b/>'
SELECT @x.query('if (/a[1]) then "true" else "false"')--返回 false
SELECT @x.query('if (/b[1]) then "true" else "false"')--返回 true
```

对于返回单个布尔值、节点序列或空序列的表达式，可以计算有效的布尔值。在处理下列类型的表达式时，将隐式计算布尔值：

- 逻辑表达式；
- NOT()函数；
- FLWOR 表达式的 WHERE 子句；
- 条件表达式；
- 量化表达式。

5．类型

XQuery 对于架构类型是强类型语言，对于非类型化的数据是弱类型语言。XQuery 中预定义的类型包括：

- http://www.w3.org/2001/XMLSchema 命名空间中 XML 架构的内置类型。
- http://www.w3.org/2004/07/xpath-datatypes 命名空间中定义的类型。

XML 架构的内置类型具有预定义的命名空间前缀 xs，例如 xs:integer 和 xs:string，所有这些内置类型都支持。可以在创建 XML 架构集合时使用这些类型。当查询类型化的 XML 时，节点的静态和动态类型由与正被查询的列或变量相关联的 XML 构架集合确定。

在 http://www.w3.org/2004/07/xpath-datatypes 命名空间中定义的类型，具有预定义的前缀 xdt。这些类型的限制条件如下：

- 在创建 XML 架构集合时无法使用这些类型。这些类型在 XQuery 类型系统中用于 XQuery 与静态类型化。可以在 xdt 命名空间中将其转换为原子类型，如 xdt:untypedAtomic。
- 查询非类型化的 XML 时，元素节点的静态类型和动态类型都是 xdt:untyped，属性值的类型是 xdt:untypedAtomic。query()方法的结果将生成非类型化的 XML。这意味着 XML 节点将分别作为 xdt:untyped 和 xdt:untypedAtomic 返回。
- 不支持 xdt:dayTimeDuration 和 xdt:yearMonthDuration 类型。

每个节点都带有类型化值和字符串值。对于类型化的 XML 数据，类型化值的类型是与正被查询的列或变量相关联的 XML 架构集合提供的。对于非类型化的 XML 数据，类型化值的类型是 xdt:untypedAtomic。

可以使用 data()或 string()函数检索节点的值：

- data()函数(XQuery)返回节点的类型化值。
- string()函数(XQuery)返回节点的字符串值。

例如代码 10.49 所示，第一个查询中表达式首先检索/root[1]的类型化值，然后向其添加 3，得到结果 8。第二个查询表达式将失败，因为表达式中的 string(/root[1])返回字符串类型值。然后此值将传递给只接受数值类型值作为操作数的算术运算符。

代码 10.49　data()函数和 string()函数的使用

```
DECLARE @x XML
SET @x='<root>5</root>'
SELECT @x.query('data(/root[1]) + 3')--返回结果 8
--以下查询失败，因为字符串类型和数值类型不能用+
SELECT @x.query('string(/root[1]) + 3')
```

6．错误处理

W3C 规范允许静态或动态引发类型错误，并定义静态错误、动态错误和类型错误。编译和错误处理是指语法不正确的 XQuery 表达式和 XML DML 语句会返回编译错误。编译阶段会检查 XQuery 表达式和 DML 语句的静态类型正确性，并针对类型化的 XML 使用 XML 架构进行类型推理。

如果表达式在运行时由于类型安全冲突而失败，会引起静态类型错误。静态错误的示

例包括将字符串添加到整数，以及在不存在的节点中查询类型化的数据。

说明：与 W3C 标准有所不同的是，XQuery 运行时错误被转换为空序列。这些序列根据调用上下文，可以作为空 XML 或 NULL 传播到查询结果。

通过显式转换为正确的类型，用户可以解决静态错误的问题，尽管运行时转换错误将被转换为空序列。静态错误是通过使用 Transact-SQL 错误机制返回的。在 SQL Server 中，XQuery 类型错误是静态返回的。

在 XQuery 中，大部分动态错误都映射到一个空序列（即“()”）。不过，有两个例外：XQuery 聚合函数中的溢出条件和 XML-DML 验证错误。另外，使用 XML 索引的查询执行可能引发错误。因此，为了能够有效地执行索引而不生成意外错误，SQL Server 数据库引擎会将动态错误映射到()。

通常，在 XQuery 内出现动态错误的情况下，不引发错误就不会更改语义，因为()映射到 False。但是，在某些情况下，返回()而不返回动态错误可能导致意外结果。例如代码 10.50 所示，对于“<c>Hello</c>”，由于是字符串，所以不可能转换为数值，那么在使用 avg() 函数求平均值时将会出现错误。在此出现的动态错误将映射到一个空序列，所以在计算平均值时，只有 100 和 200 两个数字参与运算，最后系统返回结果 150。

代码 10.50　动态错误映射到空序列

```
DECLARE @x XML
SET @x=N'<root XMLns:myNS="test">
 <a>100</a>
 <b>200</b>
 <c>Hello</c>
</root>'
SELECT @x.query('avg(//*)') --有误
```

7. 注释

可以向 XQuery 添加注释。注释字符串通过使用“(:”和“:)”分隔符来添加。在 XQuery 中添加注释的示例如代码 10.51 所示。

代码 10.51　在 XQuery 中使用注释

```
declare @x XML
set @x=''
SELECT @x.query('
(: 这里是注释:)
<ProductModel ProductModelID="10" />
')
```

10.7.2　FLWOR 语句

XQuery 中定义了 FLWOR 迭代语法，FLWOR 是 for、let、where、order by 和 return 的缩写词。FLWOR 语句由以下几个部分组成：

- 用于将一个或多个迭代器变量绑定到输入序列的一个或多个 FOR 子句。

- 输入序列可以是其他 XQuery 表达式，例如 XPath 表达式。它们既可以是节点序列也可以是原子值序列。原子值序列可以使用文字或构造函数来构造。在 SQL Server 中，不允许使用 XML 构造节点作为输入序列。
- 可选的 let 子句。该子句将一个值分配给用于特定迭代的给定变量。分配的表达式可以为 XQuery 表达式（如 XPath 表达式），并可返回一个节点序列或一个原子值序列。原子值序列可以通过使用文字或构造函数来构造。在 SQL Server 中，不允许使用 XML 构造节点作为输入序列。
- 迭代器变量。通过使用 as 关键字，此变量可包含一个可选的类型判断。
- 可选的 where 子句。此子句将对迭代应用筛选条件。
- 可选的 order by 子句。
- return 表达式。return 子句中的表达式用于构造 FLWOR 语句的结果。

下面分别举例说明 FLWOR 语句的用法。

1．for语句

XQuery 中的 for 语句与 C 语言中的 for 语句类似，用于迭代集合中的每一个值。例如代码 10.52 所示，for 语句中声明了迭代变量$step，$step 迭代器变量表示第一个 Location 节点下的所有 Step 节点。

代码 10.52　在 XQuery 中使用 for 语句

```
declare @x XML
set @x='<ManuInstructions ProductModelID="1" ProductModelName=
"SomeBike" >
<Location LocationID="L1" >
  <Step>1.1</Step>
  <Step>1.2</Step>
  <Step>1.3</Step>
</Location>
<Location LocationID="L2" >
  <Step>2.1</Step>
  <Step>2.2</Step>
  <Step>2.3</Step>
</Location>
</ManuInstructions>'        --初始化数据
SELECT @x.query('
   for $step in /ManuInstructions/Location[1]/Step
   return string($step)
') --循环实现
```

迭代 Step 节点后返回结果：

```
1.1 1.2 1.3
```

2．let语句

可以使用 let 子句来命名通过引用变量来引用的重复表达式。在 SQL Server 2012 中，每次在查询中引用 let 变量时，分配给该变量的表达式都将插入该查询中。也就是说，此语句不止执行一次，而是引用该表达式多少次，此语句就执行多少次。

在 AdventureWorks2012 数据库中，生产说明包含有关所需的工具及在何处使用这些工具的信息。如代码 10.53 中的查询使用 let 子句列出了构建生产模型所需的工具及各工具的使用位置。

代码 10.53　在 XQuery 中使用 let 语句

```
SELECT Instructions.query('
   declare namespace AWMI="http://schemas.microsoft.com/sqlserver/2004
   /07/adventure-works/
ProductModelManuInstructions";
        for $T in //AWMI:tool
            let $L := //AWMI:Location[.//AWMI:tool[.=data($T)]]
        return
          <tool desc="{data($T)}" Locations="{data($L/@LocationID)}"/>
') as Result --使用 let 语句做循环
FROM Production.ProductModel
where ProductModelID=7
```

系统返回结果如下：

```
<tool desc="T-85A framing tool" Locations="10" />
<tool desc="Trim Jig TJ-26" Locations="10" />
<tool desc="router with a carbide tip 15" Locations="10" />
<tool desc="Forming Tool FT-15" Locations="10" />
<tool desc="standard debur tool" Locations="30" />
<tool desc="paint harness" Locations="45" />
```

3．where语句

可以使用 where 子句来筛选迭代结果。例如在生产自行车时，生产过程经过了一系列生产车间。每个生产车间定义一个生产步骤序列。如代码 10.54 的查询仅检索那些生产某个自行车型号并且生产步骤少于三步的生产车间，即它们包含的<step>元素少于三个。

代码 10.54　在 XQuery 中使用 where 语句

```
SELECT Instructions.query('
   declare namespace AWMI="http://schemas.microsoft.com/sqlserver/
   2004/07/adventure-works/
ProductModelManuInstructions";
for $WC in /AWMI:root/AWMI:Location
      where count($WC/AWMI:step) < 3
      return
          <Location >
           { $WC/@LocationID }
          </Location>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

where 子句中表达式的结果使用下列规则，按指定的顺序转换为布尔值。这些规则与路径表达式中的条件相同，但不允许使用整数：

- 如果 where 表达式返回一个空序列，则其有效的布尔值为 False。
- 如果 where 表达式返回一个简单的布尔类型值，则该值为有效的布尔值。
- 如果 where 表达式返回至少包含一个节点的序列，则有效的布尔值为 True。

- 否则，它将出现静态错误。

4．order by语句

通过使用 FLWOR 表达式中的 order by 子句，可在 XQuery 中进行排序。传递到 order by 子句的排序表达式必须返回其类型对于 gt 运算符有效的值。每个排序表达式必须针对每一项生成一个单独的序列。默认情况下，按升序进行排序，也可以选择为每个排序表达式指定升序或降序顺序。

注意：在 SQL Server 中实现 XQuery 进行的字符串值的排序比较，始终是使用二进制 Unicode 码位排序规则来执行的。

使用 order by 语句时有如下限制：

- 排序表达式必须经过同类类型化，这是通过静态检查来确定的。
- 无法控制对空序列的排序。
- 不支持对 order by 使用 empty least、empty greatest 和 collation 关键字。

代码 10.55 就是对一个 XML 中的所有 Person 节点的 Name 属性进行排序。

代码 10.55　在 XQuery 中使用 order by 语句

```
declare @x XML
set @x='<root>
  <Person Name="A" />
  <Person />
  <Person Name="B" />
</root>
'
select @x.query('
  for $person in //Person
  order by $person/@Name
  return   $person
')
```

系统返回结果：

```
<Person />
<Person Name="A" />
<Person Name="B" />
```

注意：排序表达式返回空值的节点将被列在序列的开头。

10.7.3　XQuery 条件表达式

XQuery 支持以下的 if-then-else 条件语句，其语法格式如代码 10.56 所示。

代码 10.56　XQuery 条件表达式的语法

```
if (<expression1>)
then
  <expression2>
else
  <expression3>
```

与 C 语言中的 if 语句相同，系统根据 expression1 的有效布尔值，选择计算 expression2 或 expression3，但需要注意的是：

- 测试表达式必须用括号括起来。
- else 表达式是必需的。如果不需要该表达式，可以返回“()”。

代码 10.57 判断 if 语句中的表达式 sql:variable("@v")="FirstName" 为 true，返回 FirstName 节点。

代码 10.57　使用 XQuery 条件表达式

```
declare @x XML
declare @v varchar(20)
set @v='FirstName'
set @x='
<ROOT rootID="2">
  <FirstName>fname</FirstName>
  <LastName>lname</LastName>
</ROOT>'
--以下使用 XQuery 进行查询
SELECT @x.query('
if ( sql:variable("@v")="FirstName" ) then
  //FirstName
 else
   //LastName
')
```

10.7.4　XQuery 运算符

XQuery 支持下列运算符：

- 数字运算符（+、–、*、div、mod）。
- 值比较运算符（eq、ne、lt、gt、le、ge）。
- 一般比较运算符（=、!=、<、>、<=、>=）。

这里需要特别说明的是值比较运算符，由于使用的是英文缩写，所以可能很多读者不知道其中的意思。值比较运算符的说明如表 10.2 所示。

表 10.2　值比较运算符

运　算　符	说　　明	运　算　符	说　　明
Eq	等于	Gt	大于
Ne	不等于	Le	小于等于
Lt	小于	Ge	大于等于

值比较运算符用于比较原子值。在查询中可以使用常规比较运算符来代替值比较运算符。如果所选的运算符两个值的比较结果相同，则表达式将返回 true；否则将返回 false。如果其中任何一个值是空序列，则表达式的结果为 false。这些运算符只适用于单一原子值。也就是说，不能将一个序列指定为其中一个操作数。如代码 10.58 所示，通过 for 语句迭代

出每一个 Picture 节点，并且要求节点的 Size 属性等于“small”值。

代码 10.58　使用 XQuery 运算符

```
SELECT CatalogDescription.query('
            declare namespace PD="http://schemas.microsoft.com/sqlserver/
2004/07/adventure-works/ProductModelDescription";
            for $P in /PD:ProductDescription/PD:Picture[PD:Size eq "small"]
            return
                 $P
           ') as Result    --XQuery 运算符的使用
FROM Production.ProductModel
WHERE ProductModelID=19
```

系统返回结果：

```
<PD:Picture
XMLns:PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-work
s/ProductModelDescription">
  <PD:Angle>front</PD:Angle>
  <PD:Size>small</PD:Size>
  <PD:ProductPhotoID>118</PD:ProductPhotoID>
</PD:Picture>
```

需要注意在上述查询中：

- declare namespace 定义在随后的查询中使用的命名空间前缀。
- <Size>元素值与指定的原子值“small”进行比较。
- 由于值比较运算符只适用于原子值，因此隐式使用 data()函数检索节点值。也就是说，data($P/PD:Size) eq "small"生成相同的结果。

10.7.5　XQuery 函数

XQuery 函数属于 http://www.w3.org/2004/07/xpath-functions 命名空间。W3C 规范使用“fn:”命名空间前缀说明这些函数。使用这些函数时，不必显式指定“fn:”命名空间前缀。由于这个原因，也为了提高可读性，通常不使用命名空间前缀。XQuery 含有超过 100 个内建的函数。这些函数可用于字符串值、数值、日期及时间比较、节点和 QName 操作、序列操作、逻辑值等。

前面例子中使用到的 sum()、avg()函数等都是 XQuery 内建函数，例如要查询一个书籍列表中最便宜的数据的价格，则对应的脚本如代码 10.59 所示。

代码 10.59　使用 XQuery 函数

```
DECLARE @books XML
SET @books='
<books>
  <book>
    <name>大学物理</name>
    <price>20</price>
  </book>
  <book>
    <name>高等数学</name>
    <price>30</price>
  </book>
```

```
</books>
'
SELECT @books.query('min(//book/price/text())')  --使用函数
```

10.8　小　　结

本章主要讲解了 XML 的基础知识和 SQL Server 和 XML 的结合使用。SQL Server 2000 中已经提供了 FOR XML 语句用于将查询结果生成 XML。在 SQL Server 2005 中加大了对 XML 的支持，提供了 XML 数据类型和对应的 XML 操作的方法。

FOR XML 子句用于查询中将查询结果生成 XML 数据。SQL Server 为 FOR XML 子句提供了 4 种模式，分别是 RAW、AUTO、EXPLICIT 和 PATH。

XML 数据类型可以用于表的列、变量和参数。使用 query()、exists()、value()、modify()和 nodes()方法可以自己在 SQL 语句中操作 XML。为了提高 XML 数据列的查询性能，SQL Server 提供了 XML 索引支持。XML 索引分为 XML 主索引和 XML 辅助索引。

在对 XML 数据进行查询和处理的过程中，XQuery 起着相当重要的作用。XQuery 被设计用来查询 XML 数据。XQuery 相对于 XML，等同于 SQL 相对于数据库。XQuery 提供了 FLWOR 语句、条件表达式、运算符和函数等用于完成复杂的 XML 数据处理操作。

第 11 章　使用 ADO.NET

ADO.NET 是一组向.NET 程序员公开数据访问服务的类。ADO.NET 为创建分布式数据共享应用程序提供了一组丰富的组件，提供了对关系数据、XML 和应用程序数据的访问。本章将主要讲解 ADO.NET 的使用。

11.1　ADO.NET 概述

ADO.NET 与.NET Framework 一起推出，随着.NET Framework 的更新和强大，ADO.NET 的功能也不断地发展和强大。本节将主要介绍 ADO.NET 的基础知识，使读者对 ADO.NET 有一个基础的了解。

11.1.1　ADO.NET 发展历史

ADO.NET 在 2003 年推出时，内建在.NET Framework 1.0 之中，当时的版本为 1.0。在 2005 年.NET Framework 2.0 版推出时，内建了 ADO.NET 2.0 版。相对 ADO.NET 1.0 版本，2.0 中添加了对分布式事务的支持，另外还添加了对 SQL 2005 新数据类型 XML 的支持。

在 2006 年年底微软推出了.NET Framework 3.0 版本，在此版本全面提供了更好的服务导向（SOA）基础技术、更生动活泼的交互式应用程序接口，以及更严谨、安全的网络通信架构和企业商业流程的引擎。.NET Framework 3.0 是一个附加在.NET Framework 2.0 架构之上，让程序开发者在设计应用程序的过程中能更加得心应手的链接库，其中主要包含了以下几大部分。

- Windows Presentation Foundation（WPF）：以新的应用程序架构使程序开发人员能够设计出具有视觉效果且画面很炫的程序接口。
- Windows Communication Foundation（WCF）：有安全通信能力的应用程序。
- Windows Workflow Foundation（原来叫 WWF，由于重名所以后改为 WF）：整合商业流程的应用程序。
- Windows Card Space（WCS）：代表在不同的情境下，个人的身份识别。

.NET Framework 2.0 提供的功能并没有丝毫变动，也就是说，关于 ADO.NET 2.0 提供的功能是完全一样的，版本命名也沿用 2.0 版。

2007 年微软推出了.NET Framework 3.5 版。.NET Framework 3.5 是以.NET Framework 2.0 为基础，整合.NET Framework 3.0 版提供的功能，再外加上新一代的 LINQ、ASP.NET 3.5 与其他各种服务，它们之间的关系，如图 11.1 所示。

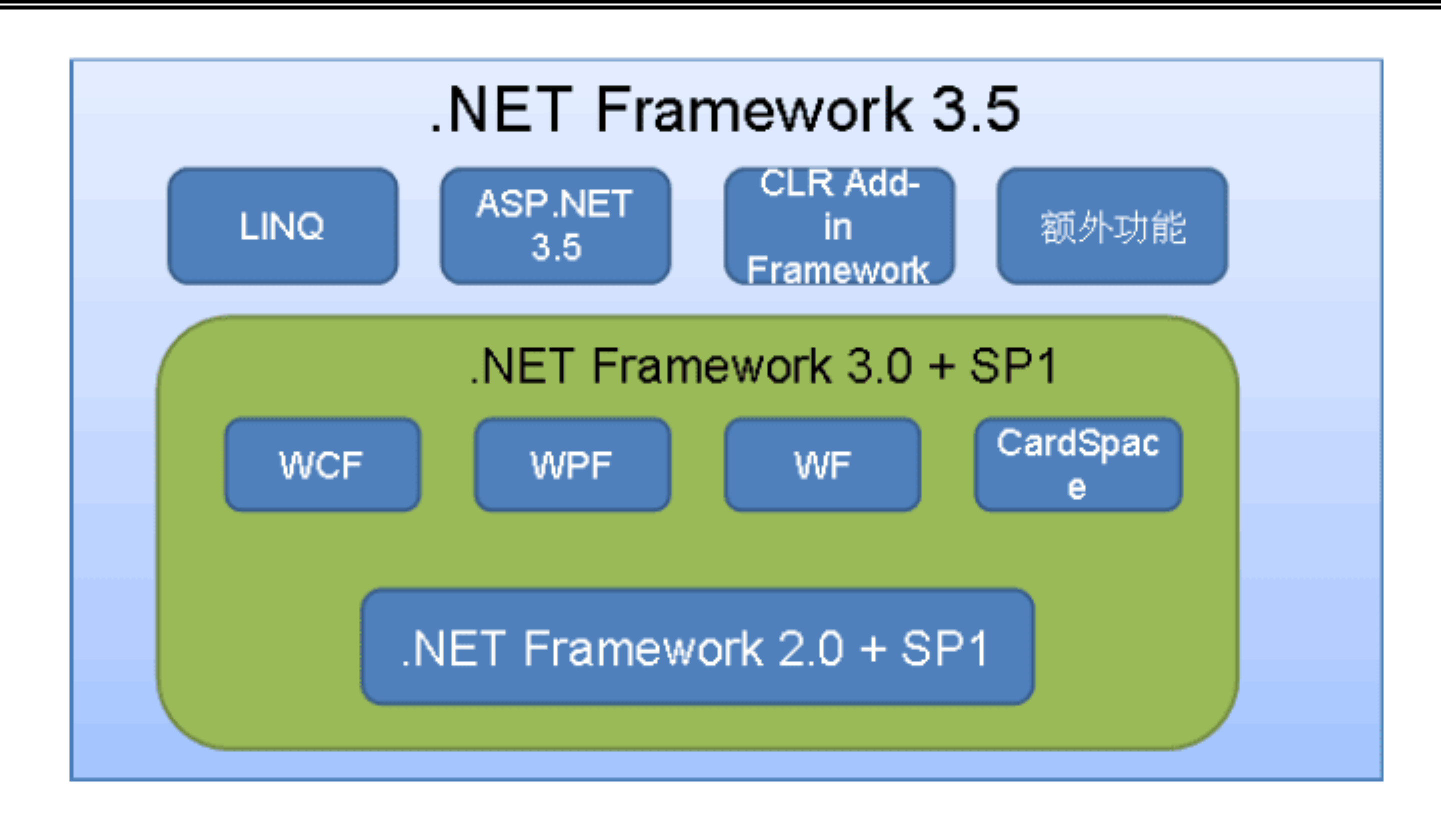


图 11.1　.NET Framework 3.5 包含的内容

从图 11.1 中可看出，在 VS 2008 开发工具与.NET Framework 3.5 版发行时，ADO.NET 核心功能部分并没有变动，还是以 ADO.NET 2.0 为基础。可以从 GAC（预设是 C:\Windows\Assembly 目录）中检视 ADO.NET 核心类别库 System.Data.dll 的版本，它还是 2.0 版。但.NET Framework 3.5 多了一些与 ADO.NET 相关的扩充函数库，如 System.Data.DataExtensions.dll（3.5 版）、System.Data.Linq.dll（3.5 版），提供额外的数据存取功能。换句话说，ADO.NET 2.0 的程序可以毫无问题地正确移转到.NET Framework 3.5 的平台上执行，也能利用新的类库带来的好处。

.NET Framework 3.5 版所提供的 ADO.NET 称为 ADO.NET 3.5。在 ADO.NET 3.5 版本正式发行之前，ADO.NET 有个 3.0 的测试版，但在.NET Framework 3.5 正式发行时，并不包含 ADO.NET 3.0 测试版中主张要提供的 ADO.NET Entity Framework，也不提供 Microsoft Synchronization Services for ADO.NET。ADO.NET Entity Framework 主要的功能是，让程序设计员能够透过一个对应到数据结构描述信息的 Entity 数据模型的概念层，很容易地操作数据。

Microsoft Synchronization Services for ADO.NET 是一个开发架构，着重在 2 层式、N-层式应用程序（即分布式应用程序），以及服务导向程序之间的数据同步动作。它提供一组 API 接口和组件，让行动装置应用程序、数据服务与本机数据储存体能同步保持一致，以设计更完善的离线应用程序。

除此之外，ADO.NET 3.5 版本中也提供许多支持 SQL Server 2008 的新功能，像新的 date、time 数据类型的支持等。

2010 年微软推出了.NET Framework 4.0 版，对应的开发平台是 VS 2010。此版本在之前版本的基础上做了大量的改进。ADO.NET 4.0 也同时得到了改进，主要包括如下几方面。

- 可以在创建概念模型时，使数据库中的外键列对应于实体类型的标量属性。
- 可以在使用 N 层应用程序时使用自跟踪实体。自跟踪实体可记录对标量属性、复杂属性和导航属性所做的更改。自跟踪对象中的跟踪信息可以在服务端应用于对象上下文。
- 新增加了用于 N 层应用程序开发的新方法。
- EntityDataSource 控件支持 QueryExtender 控件，主要用来为从数据源检索的数据创建筛选器。查询扩展程序控件可以通过使用声明性语法筛选网页标记中的数据。
- 新增了 ADO.NET 实体数据模型设计器。它是支持通过单击鼠标修改.edmx 文件的

工具。使用实体设计器可以直观地创建和修改实体、关联、映射和继承关系，同时还能够验证.edmx 文件。

此外，还在 LINQ 方面做了较多的增强，比如：在 LINQ to Entities 查询中新增函数、以及增强 OrderBy 功能等方面。如果读者想了解更多关于 ADO.NET 4.0 的相关知识，可以在微软网站上查找。

2012 年微软推出了.NET Framework 4.5 版，对应的开发平台是 VS 2012。此版本中对 ADO.NET 中的新增功能如下。

- ❑ SqlClient 流支持。它能够支持非结构化数据，如文档、图像和媒体文件。同时无须完全将数据加载到内存，从而减少内存溢出异常。此外，还在 SqlDataReader 和 DbDataReader 中新增了功能，并更好地支持流的使用。
- ❑ ADO.NET 跟踪功能。它用于 SQL Server、Oracle、OLE DB 和 ODBC 的.NET 数据提供程序以及 ADO.NET DataSet 和 SQL Server 网络协议中。
- ❑ SqlClient 还支持 SQL Server 高可用性、灾难恢复和 AlwaysOn。
- ❑ SqlCredential 类。它由用户 ID 和将用于 SQL Server 身份验证的密码构成。SqlCredential 对象中的密码是 SecureString 类型。

此外，SqlClient 对稀疏列提供额外的支持，并且还支持 LocalDb 数据库连接的操作。目前，微软即将推出.NET 5.0 框架，感兴趣的读者可以关注微软网站。

11.1.2　ADO.NET 的结构

可以使用 ADO.NET 的以下两个组件访问和处理数据：

- ❑ .NET Framework 数据提供程序。
- ❑ DataSet。

.NET Framework 数据提供程序是专门为数据处理及快速地只进、只读访问数据而设计的组件。Connection 对象提供与数据源的连接。Command 对象使程序员能够访问用于返回数据、修改数据、运行存储过程及发送或检索参数信息的数据库命令。DataReader 从数据源中提供高性能的数据流。最后，DataAdapter 提供连接 DataSet 对象和数据源的桥梁。DataAdapter 使用 Command 对象在数据源中执行 SQL 命令，以便将数据加载到 DataSet 中，并使对 DataSet 中数据的更改与数据源保持一致。

DataSet 是专门为独立于任何数据源的数据访问而设计。因此，它可以用于多种不同的数据源，用于 XML 数据或用于管理应用程序本地的数据。DataSet 包含一个或多个 DataTable 对象的集合，这些对象由数据行和数据列及有关 DataTable 对象中数据的主键、外键、约束和关系信息组成。如图 11.2 所示为.NET Framework 数据提供程序与 DataSet 之间的关系。

说明：.NET Framework 数据提供程序是联机访问数据，DataSet 是脱机访问数据。DataSet 一次性将数据载入到内存中，访问 DataSet 时只访问内存，而不会再访问数据库。

在数据库开发中使用 DataReader 和 DataSet 都可以很好地实现功能，但是在以下情况中更适合使用 DataSet。

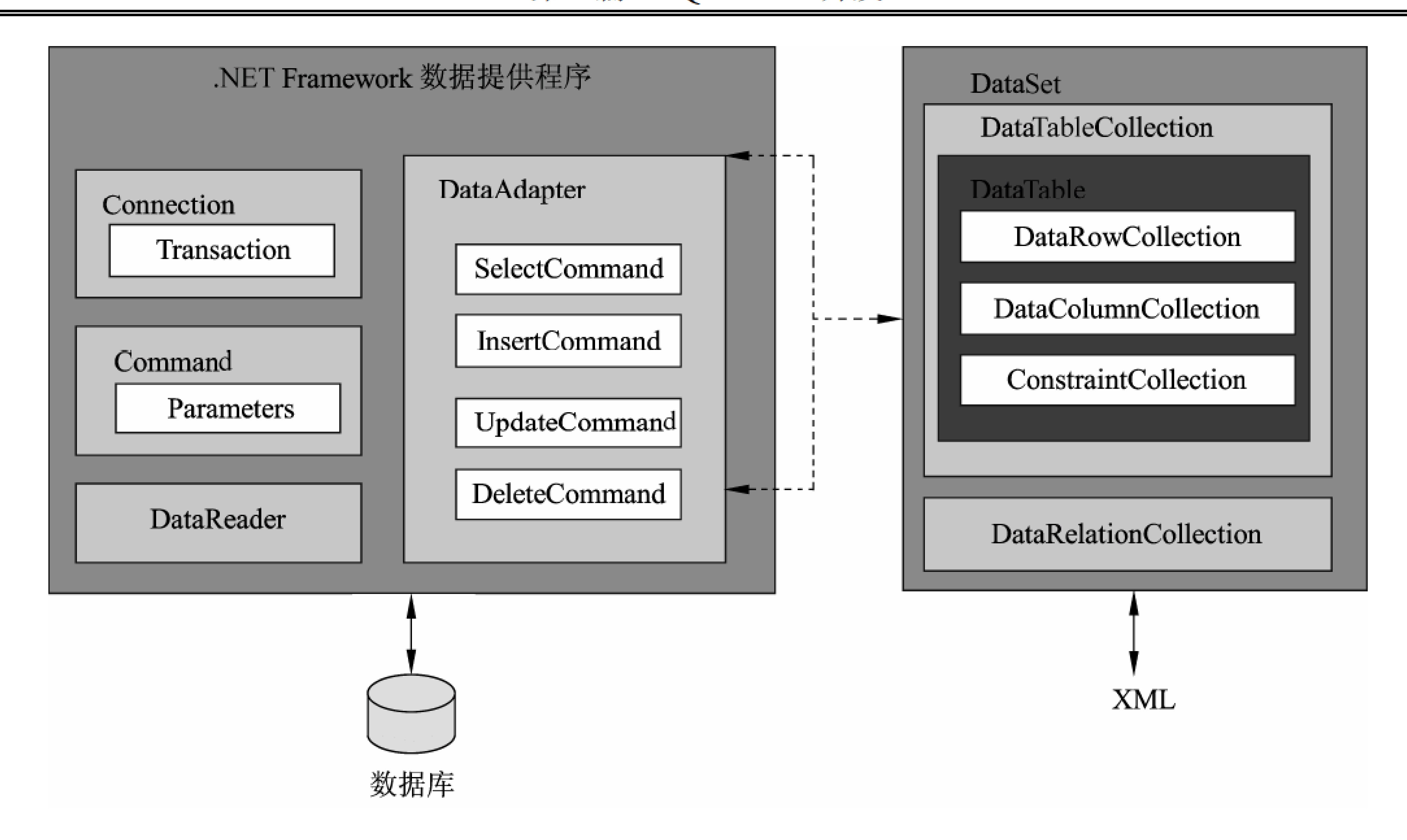


图 11.2　ADO.NET 的两个组件

- ❑ 在应用程序中将数据缓存在本地，以便可以对数据进行处理。如果只需要读取查询结果，DataReader 是更好的选择。
- ❑ 在层间或从 XMLWeb 服务对数据进行远程处理。
- ❑ 与数据进行动态交互，例如绑定到 Windows 窗体控件或组合并关联来自多个源的数据。
- ❑ 对数据执行大量的处理，而不需要与数据源保持打开的连接，从而将该连接释放给其他客户端使用。

如果不需要 DataSet 所提供的功能，则可以使用 DataReader 以只进、只读方式返回数据，从而提高应用程序的性能。DataAdapter 设计使用 DataReader 填充 DataSet 的内容，实现 DataReader 到 DataSet 的转换。但由于 DataSet 是将数据保存在内存中，所以使用 DataReader 来提高性能，因为这样可以节省 DataSet 所使用的内存，并省去了创建 DataSet 及填充其内容所需的处理。

注意：ADO.NET 的结构表明，ADO.NET 并不是只为 SQL Server 的数据访问而设计的，无论是 XML 数据源还是 Oracle 作为数据源，都可以通过 ADO.NET 进行访问。

11.1.3　ADO.NET 的优点

随着技术的发展，现在的应用程序需要软件具有断开式数据结构；能够与 XML 紧密集成；具有能够组合来自多个不同数据源数据的通用数据表示形式；以及具有为与数据库交互而优化的功能。而 ADO.NET 具有如下优点：

- ❑ 利用当前的 ActiveX 数据类型（ADO）知识。
- ❑ 支持 N 层编程模型。
- ❑ 集成 XML 支持。

ADO.NET 的设计满足了当今应用程序开发模型的多种要求。同时，该编程模型尽可能地与 ADO 保持一致，使现在的 ADO 开发人员不必从头开始学习。ADO.NET 是.NET Framework 的固有部分，ADO 程序员仍很熟悉。

ADO.NET 还与 ADO 共存。虽然大多数基于.NET 的新应用程序将使用 ADO.NET 来编写，但.NET 程序员仍然可以通过.NETCOM 互操作性服务来使用 ADO。

使用断开式数据集这一概念已成为编程模型中的焦点。ADO.NET 为断开式 N 层编程环境提供了一流的支持，许多新的应用程序都是为该环境编写的。N 层编程的 ADO.NET 解决方案就是 DataSet。

XML 和数据访问紧密联系在一起。XML 与编码数据有关，数据访问也越来越多地与 XML 有关。.NET Framework 不仅支持 Web 标准，还是完全基于 Web 标准生成的。

XML 支持内置在 ADO.NET 中非常基本的级别上。.NET Framework 和 ADO.NET 中的 XML 类是同一结构的一部分，它们在许多不同的级别集成。因此不必在数据访问服务集和它们的 XML 相应服务之间进行选择，它们的设计本来就具有从其中一个跨越到另一个的功能。

11.2　建立与管理连接

ADO.NET 中执行数据处理的第一步就是建立与数据库的连接，建立到 SQL Server 2012 的连接使用 SqlConnection 对象。本章将主要讲解连接的基础知识。

11.2.1　连接字符串

连接字符串的格式是使用分号分隔的键/值参数对列表：

```
keyword1=value; keyword2=value
```

连接字符串忽略空格，关键字不区分大小写，尽管值可能会区分大小写，这取决于数据源的大小写。如果要加入包含分号、单引号或双引号的值，则值必须加双引号。

1．持续安全信息

连接字符串中 Persist Security Info 关键字表示持续安全信息，默认设置为 false。

如果将该关键字设置为 true 或 yes，将允许在打开连接后，从连接中获得涉及安全性的信息（包括用户标识和密码）。如果在建立连接时必须提供用户标识和密码，最安全的方法是在使用信息打开连接后丢弃这些信息，在 Persist Security Info 设置为 false 或 no 时会发生这种情况。当向不可信的源提供打开的连接，或将连接信息永久保存到磁盘时，这点尤其重要。如果将 Persist Security Info 保持为 false，可帮助确保不可信的源无法访问连接中涉及安全性的信息，并帮助确保任何涉及安全性的信息都不会随连接字符串信息在磁盘上持久化。

2．Windows身份验证

在连接字符串中使用 Integrated Security 关键字，来表示是否使用基于 Windows 的身份

认证。如果设置为 true，则表示启用 Windows 身份认证，连接字符串如下：

```
Integrated Security=true;
```

如果是使用 OleDb .NET Framework 数据提供程序，则使用 Windows 身份认证的配置字符串为：

```
Integrated Security=SSPI;
```

对于 ODBC .NET Framework 数据提供程序，必须使用以下键/值对指定 Windows 身份验证：

```
Trusted_Connection=yes;
```

对于 SQL Server 2012，由于使用 OLE DB 进行访问，所以若使用 Windows 身份认证时，配置为：

```
Integrated Security=SSPI;
```

3．服务器和数据库

在连接字符串中服务器可以使用 Data Source 关键字来指定。服务器若在局域网中可以使用服务器名也可以使用 IP，若在互联网中则可以使用域名或者 IP，若客户端配置了服务器的别名则可以使用服务器别名。如果连接的数据库实例不是默认实例，则还需要指定实例名，例如：

```
Data Source=IBM-PC\MSSQLSERVER
```

服务器名除了使用 Data Source 关键字外也可以使用 Server 关键字来表示。使用方法相同，例如：

```
Server=IBM-PC\MSSQLSERVER
```

需要连接的数据库使用 Initial Catalog 关键字来表示。例如要连接到 TestDB1 数据库，则对应的连接字符串为：

```
Initial Catalog= TestDB1
```

除此之外还可以使用 Database 关键字来表示，例如：

```
Database= TestDB1
```

4．用户名密码

对于 Windows 身份认证不需要再提供用户名和密码，但是如果使用 SQL 身份认证，则需要在连接字符串中声明用户名和密码。

声明用户名使用 User ID 关键字或者使用 Uid 关键字来表示。而声明密码使用 Password 关键字或者简写为 Pwd 关键字来表示。例如使用 sa 用户连接到数据库，连接的密码是 p@ssw0rd，则对应的连接字符串为：

```
User ID=sa;Password=p@ssw0rd;
```

假设现在需要连接到数据库服务器 IBM-PC，连接的数据库是 TestDB1，使用 SQL 身份认证方式连接，则完整的数据库连接字符串为：

```
server=IBM-PC;database=TestDB1;uid=sa;pwd=p@ssw0rd
```

技巧： 对于不同的数据库有不同的连接字符串，读者不需要对每种数据库的连接字符串都倒背如流。在不知道连接字符串格式的情况下可以新建一个后缀为 udl 的文件，双击打开可以用可视化的方式设置要连接到的数据库，设置好后保存，然后用文本编辑器打开该文件即可看到完整的数据库连接字符串了。

11.2.2　建立和断开连接

ADO.NET 中使用 SqlConnection 对象建立和断开连接，在构造 SqlConnection 对象实例时便可指定连接字符串。构造 SqlConnection 对象的 C#代码如代码 11.1 所示。

代码 11.1　构造 SqlConnection 对象

```
string sqlconn = "server=IBM-PC;database=TestDB1;uid=sa;pwd=p@ssw0rd";
SqlConnection conn = new SqlConnection(sqlconn);
```

SqlConnection 实例提供了 Open()方法用于打开该连接，只有打开数据库连接后才能对数据库进行操作。

```
conn.Open();
```

执行数据库操作后如果没有销毁 SqlConnection 对象或者没有显式关闭该连接，则数据库连接将不会自动关闭。关键数据库连接使用 Close()方法：

```
conn.Close();
```

由于数据库连接是稀缺资源，所以如果大量打开的连接没有关闭，将会造成系统缓慢甚至抛出异常。所以针对数据库连接的原则是尽量晚些打开，尽量早些关闭。

使用 SqlConnection 打开连接、执行数据库操作，最后关闭数据库连接的编程方式有两种。一种是使用 using 语句，通过回收 SqlConnection 对象的方式来关闭数据库连接，如代码 11.2 所示。

代码 11.2　使用 using 来建立和断开连接

```
string sqlconn = "server=IBM-PC;database=TestDB1;uid=sa;pwd=p@ssw0rd";
//连接字符串
using( SqlConnection conn = new SqlConnection(sqlconn))
{
    conn.Open();
    //数据库操作
}//结束时将自动断开连接
```

另外一种方式就是在 finally 语句块中执行数据库连接的关闭。因为不管数据库操作中是否发生了异常，数据库连接都能够关闭。具体 C#代码如代码 11.3 所示。

代码 11.3　在 finally 中关闭数据库连接

```
string sqlconn = "server=IBM-PC;database=TestDB1;uid=sa;pwd=p@ssw0rd";
//连接字符串
SqlConnection conn = new SqlConnection(sqlconn);//定义连接对象
```

```
try{
    conn.Open();
    //数据库操作
}
catch{
    //异常处理
}
finally{
    if (conn != null) {//没有关闭连接的话就关闭连接
        conn.Close();
    }
}
```

11.2.3　数据库连接池概述

连接到数据库服务器通常由几个需要很长时间的步骤组成。必须建立物理通道（例如套接字或命名管道）；必须与服务器进行初次握手；必须分析连接字符串信息；必须由服务器对连接进行身份验证；必须运行检查以便在当前事务中登记等。

实际上，大多数应用程序仅使用一个或几个不同的连接配置。这意味着在执行应用程序期间，许多相同的连接将被反复地打开和关闭。为了使打开的连接成本最低，ADO.NET 使用称为连接池的优化方法。

连接池减少新连接需要打开的次数并保持物理连接的所有权。通过为每个给定的连接配置保留一组活动连接来管理连接。只要用户在连接上调用 Open，池进程就会检查池中是否有可用的连接。如果某个池连接可用，会将该连接返回给调用者，而不是打开新连接。应用程序在该连接上调用 Close 时，池进程会将连接返回到活动连接池集中，而不是真正关闭连接。连接返回到池中之后，即可在下一个 Open 调用中重复使用。

只有配置相同的连接可以建立池连接。ADO.NET 同时保留多个池，每个配置对应一个池。连接由连接字符串及 Windows 标识（在使用集成的安全性时）分为多个池。

池连接可以大大提高应用程序的性能和可缩放性。默认情况下，ADO.NET 中启用连接池。除非显式禁用，否则，连接在应用程序中打开和关闭时，池进程将对连接进行优化。还可以提供几个连接字符串修饰符来控制连接池的行为。

11.2.4　创建连接池

在初次打开连接时，将根据完全匹配算法创建连接池，该算法将池与连接中的连接字符串关联。每个连接池与不同的连接字符串关联。打开新连接时，如果连接字符串并非与现有池完全匹配，将创建一个新池。按进程、应用程序域、连接字符串以及（在使用集成的安全性时）Windows 标识来建立池连接。

如果 MinPoolSize 在连接字符串中未指定或指定为 0，则池中的连接将在一段时间不活动后关闭。但是，如果指定的 MinPoolSize 大于 0，在 AppDomain 被卸载并且进程结束之前，连接池不会被破坏。非活动或空池的维护只需要最少的系统开销。

11.2.5　添加连接

连接池是为每个唯一的连接字符串创建的。当创建一个池后，将创建多个连接对象并

将其添加到该池中，以满足最小池大小的要求。连接根据需要添加到池中，但是不能超过指定的最大池大小（默认值为 100 个连接）。连接在关闭或断开时释放回池中。

在请求 SqlConnection 对象时，如果存在可用的连接，将从池中获取该对象。连接要可用，必须未使用，具有匹配的事务上下文或未与任何事务上下文关联，并且具有与服务器的有效连接。

连接池进程通过在连接释放回池中时重新分配连接，来满足这些连接请求。如果已达到最大池大小且不存在可用的连接，则该请求将会排队。然后，池进程尝试重新建立任何连接，直到到达超时时间（默认值为 15 秒）。如果池进程在连接超时之前无法满足请求，将引发异常。

11.2.6　移除连接

连接池进程定期扫描连接池，查找没有通过 Close 或 Dispose 关闭的未用连接，并重新建立找到的连接。如果应用程序没有显式关闭或断开其连接，连接池进程可能需要很长时间才能重新建立连接，所以，最好确保在连接中显式调用 Close 和 Dispose。

如果连接长时间空闲，或池进程检测到与服务器的连接已断开，连接池进程会将该连接从池中移除。注意，只有在尝试与服务器进行通信之后才能检测到断开的连接。如果发现某连接不再连接到服务器，则会将其标记为无效。无效连接只有在关闭或重新建立后，才会从连接池中移除。

如果存在与已消失的服务器的连接，那么即使连接池管理程序未检测到已断开的连接并将其标记为无效，仍有可能将此连接从池中取出。这种情况是因为检查连接是否仍有效的系统开销将造成与服务器的另一次往返，从而抵消了池进程的优势。发生此情况时，初次尝试使用该连接时将检测连接是否曾断开，并引发异常。

11.2.7　配置连接池

SqlConnection 对象的 ConnectionString 属性支持连接字符串键/值对，可以用于调整连接池逻辑的行为。如表 11.1 列出了用于配置连接池的连接字符串键，只需要修改连接字符串便可完成对连接池的设置。

表 11.1　连接字符串配置连接池

名　称	默 认 值	说　明
Connection Lifetime	0	当连接被返回到池时，将其创建时间与当前时间作比较，如果时间长度（以秒为单位）超出了该值，该连接就会被销毁。这在聚集配置中很有用（用于强制执行运行中的服务器和刚置于联机状态的服务器之间的负载平衡）。0 值将使池连接具有最大的连接超时
Connection Reset	'true'	确定从池中提取数据库连接时是否重置数据库连接。对于 SQL Server 7.0 版，设置为 false 可避免获取连接时再有一次额外的服务器往返行程，但须注意此时并未重置连接状态（如数据库上下文）。只要不将 Connection Reset 设置为 false，连接池程序就不会受到 ChangeDatabase()方法的影响。连接在退出相应的连接池以后将被重置，并且服务器将移回登录时的数据库。不会创建新的连接，也不会重新进行身份验证。如果将 Connection Reset 设置为 false，则池中可能会产生不同数据库的连接

续表

名　称	默 认 值	说　明
Enlist	'true'	当该值为 true 时，池程序在创建线程的当前事务上下文中自动登记连接。可识别的值为 true、false、yes 和 no
Load Balance Timeout	0	连接被销毁前在连接池中生存的最短时间（以秒为单位）
Max Pool Size	100	池中允许的最大连接数
Min Pool Size	0	池中允许的最小连接数
Pooling	'true'	当该值为 true 时，系统将从适当的池中提取 SQLConnection 对象，或在需要时创建该对象并将其添加到适当的池中。可识别的值为 true、false、yes 和 no

注意：连接池对开发人员来说是透明的，所以开发人员可以不用过多关注连接池的情况，系统将会自动完成所有的处理。在开发基于数据库的 Windows 程序时，如果 Windows 客户端较多，则最好不要使用客户端直接连接数据库的方式，因为每个客户端上都保存一个连接池，如果客户端很多，将会造成与 SQL Server 保持的连接数过多，造成系统缓慢或异常。可以采用 WCF 或者 Web 服务的方式使整个系统只存在一个连接池。

11.3　使用 SqlCommand 执行数据操作

当建立与数据源的连接后，可以使用 Command 对象执行命令并从数据源中返回结果。可以使用 Command 构造函数创建命令，该构造函数采用在数据源、Connection 对象和 Transaction 对象中执行的 SQL 语句的可选参数。也可以使用 Connection 的 CreateCommand() 方法创建用于特定连接的命令。可以使用 CommandText 属性查询和修改 Command 对象的 SQL 语句。

随.NET Framework 提供的每个.NET Framework 数据提供程序包括一个 Command 对象 OLE DB；.NET Framework 数据提供程序包括一个 OleDbCommand 对象；SQL Server .NET Framework 数据提供程序包括一个 SqlCommand 对象；ODBC .NET Framework 数据提供程序包括一个 OdbcCommand 对象；Oracle .NET Framework 数据提供程序包括一个 OracleCommand 对象。这里主要讲解针对 SQL Server 的 SqlCommand 对象。

11.3.1　构造 SqlCommand 对象

SqlCommand 对象提供了 4 个构造函数用于构造 SqlCommand 实例。无论是用哪个构造函数，构造出的 SqlCommand 实例必须要指定对应的 SqlConnection 实例。通过 SqlCommand 实例的 Connection 属性可以获得或赋予 SqlConnection 实例。

如果是执行 SQL 语句，则将 SqlCommand 实例的 CommandText 属性设置为需要执行的 SQL 语句。例如构造一个 SqlCommand 对象，并执行查询 Person.AddressType 表内容的 SQL 语句，其代码如代码 11.4 所示。

代码 11.4　构造 SqlCommand 实例执行 SQL 语句

```
string sql = "select * from Person.AddressType";//定义 SQL 语句字符串
SqlCommand cmd = new SqlCommand(sql, conn);     //声明一个 SqlCommand 对象
```

如果要执行的是存储过程，则将 CommandText 属性设置为存储过程的名字，Parameters 属性设置为调用存储过程时使用的参数，还需要将 CommandType 属性设置为 CommandType.StoredProcedure。如要调用存储过程 GetDeparmentByGroupName，该存储过程需要一个参数@groupName，构造调用该存储过程的 SqlCommand 实例如代码 11.5 所示。

代码 11.5　构造 SqlCommand 实例执行存储过程

```
SqlCommand cmd = new SqlCommand("GetDeparmentByGroupName", conn);
//声明一个 SqlCommand 对象
cmd.CommandType = CommandType.StoredProcedure; //SqlCommand 是用于存储过程
//以下是为 SqlCommand 传入参数
cmd.Parameters.Add(new SqlParameter("@groupName", "Research and Develop
ment"));
```

存储过程除了可以接收输入参数外还可以接收输出参数。最常见的一种情况就是一个存储过程负责向一个表中插入一行数据，由于该表的主键设置为自增列，所以需要在插入完成后将主键以输出参数的方式返回给客户端。例如一个负责插入新闻的存储过程 InsertNews，其定义如代码 11.6 所示。

代码 11.6　有输出参数的存储过程

```
CREATE PROC InsertNews --创建存储过程
    @newsTitle nvarchar(100),
    @newsContent nvarchar(max),
    @newsID int out --输出参数
AS
    INSERT INTO News(Title,Content)
    values(@newsTitle,@newsContent)
    SET @newsID=@@IDENTITY
```

使用 SqlCommand 对象调用该存储过程时需要指定 3 个参数，其中输出参数的 Direction 属性为 Output 即可。对于上面的存储过程，使用 SqlCommand 调用该存储过程的代码如代码 11.7 所示。

代码 11.7　调用有输出参数的存储过程

```
SqlCommand cmd = new SqlCommand("InsertNews", conn);
//使用存储过程的名字定义 SqlCommand 对象
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue("@newsTitle", "新闻标题"); //传入存储过程参数
cmd.Parameters.AddWithValue("@newsContent", "新闻内容");
SqlParameter par = new SqlParameter("@newsID", SqlDbType.Int);
par.Direction = ParameterDirection.Output;            //设置为输出参数
cmd.Parameters.Add(par);
```

最后使用 SqlCommand 提供的方法执行该存储过程后，通过 par.Value 即可获得返回的值。

11.3.2　SqlCommand 提供的方法

SqlCommand 提供了以下几个重要的方法用于执行数据库操作。

- ExecuteReader()方法：执行返回行的命令。为了提高性能，ExecuteReader()方法使用 Transact-SQL sp_executesql 系统存储过程调用命令。因此，如果 ExecuteReader()方法用于执行命令（例如 Transact-SQL SET 语句），则它可能不会产生预期的效果。
- ExecuteNonQuery()方法：执行 T-SQL INSERT、DELETE、UPDATE 及 SET 等语句命令。
- ExecuteScalar()方法：从数据库中检索单个值（例如一个聚合值）。
- ExecuteXmlReader()方法：将 CommandText 发送到 Connection 并生成一个 XmlReader 对象。

ExecuteReader()方法返回的是一个 SqlDataReader 对象，通过该对象可以以流的形式读取数据库返回的结果。11.4 节将主要介绍 SqlDataReader 对象的使用。

ExecuteNonQuery()方法用于获得数据库受影响的行数，也就是数据库中被修改、插入或删除的行数。如果执行的是 SELECT 语句，则该函数返回–1。例如要更改 Person.AddressType 表的 ModifiedDate，则获得被更改行数的代码如代码 11.8 所示。

代码 11.8　使用 ExecuteNonQuery()方法获得更改的行数

```
string sql = "update Person.AddressType set ModifiedDate=getdate()";
//定义 SqlCommand 中的 SQL 语句
SqlCommand cmd = new SqlCommand(sql, conn);//声明 SqlCommand 对象
int count = cmd.ExecuteNonQuery();          //执行 SQL，返回更改了多少行数据
```

ExecuteScalar()方法用于返回数据库执行后的一个单值，比如执行了 SELECT COUNT(*)之类的 SQL 命令数据库将返回一行一列的数据。例如要查询 Person.AddressType 表中的行数，则对应的 C#语句如代码 11.9 所示。

代码 11.9　使用 ExecuteScalar()方法查询表的行数

```
string sql="select count(*) from Person.AddressType";//定义 SQL 语句返回的行数
SqlCommand cmd = new SqlCommand(sql, conn);
Console.WriteLine(cmd.ExecuteScalar().ToString()); //输出 SQL 返回的结果
```

ExecuteXmlReader()函数返回的是一个 XmlReader 对象，例如使用 FOR XML 子句返回一个 XML 结果，然后使用该方法读取并显示 XML 结果的程序如代码 11.10 所示。

代码 11.10　使用 ExecuteXmlReader()方法读取 XML 结果

```
string sql = "select * from Person.AddressType FOR XML RAW,ROOT('Types')";
//定义 XML 相关的 SQL 语句
SqlCommand cmd = new SqlCommand(sql, conn);
XmlReader xr= cmd.ExecuteXmlReader(); //XML 读取结果
while (xr.Read())//循环读取
{
    Console.WriteLine(xr.ReadOuterXml());//读取 XML 中的内容
}
```

除此之外，SqlCommand 实例还提供了异步执行 SQL 语句或存储过程的方法。

- BeginExecuteNonQuery()方法：启动此 SqlCommand 描述的 Transact-SQL 语句或存储过程的异步执行，一般情况下执行 INSERT、DELETE、UPDATE 和 SET 语句等命令。每调用一次 BeginExecuteNonQuery()方法，都必须调用一次通常在单独的线程上完成操作的 EndExecuteNonQuery()方法。
- BeginExecuteReader()方法：启动此 SqlCommand 描述的 Transact-SQL 语句或存储过程的异步执行，并从服务器中检索一个或多个结果集。每调用一次 BeginExecute Reader()方法，都必须调用一次通常在单独的线程上完成操作的 EndExecuteReader()方法。
- BeginExecuteXmlReader()方法：启动此 SqlCommand 描述的 Transact-SQL 语句或存储过程的异步执行。每调用一次 BeginExecuteXmlReader()方法，都必须调用一次 EndExecuteXmlReader()方法，它通常在单独的线程上完成操作，并且返回一个 XmlReader 对象。

11.4　使用 SqlDataReader 读取数据

使用 SqlCommand 实例的 ExecuteReader()方法将返回一个 SqlDataReader 对象。可以使用 DataReader 从数据库中检索只读、只进的数据流。查询结果在查询执行时返回，并存储在客户端的网络缓冲区中，直到使用 DataReader 的 Read()方法对它们发出请求。使用 DataReader 可以提高应用程序的性能，原因是只要数据可用就立即检索数据，并且（默认情况下）一次只在内存中存储一行，减少了系统开销。

11.4.1　使用 SqlDataReader 获得数据流

使用 DataReader 对象的 Read()方法可从查询结果中获取行。通过向 DataReader 传递列的名称或序号引用，可以访问返回行的每一列。不过，为了实现最佳性能，DataReader 提供了一系列方法，使程序员能够访问其本机数据类型（GetDateTime()、GetDouble()、GetGuid()、GetInt32()等）的列值。

例如要使用 DataReader 读取 Person.AddressType 表中的数据，然后将其中的 AddressTypeID 列和 Name 列输出的程序如代码 11.11 所示。

代码 11.11　使用 SqlDataReader 读取数据

```
string sql = "select * from Person.AddressType";//定义 SQL 语句返回所有 Address
                                                Type 行
SqlCommand cmd = new SqlCommand(sql, conn);
SqlDataReader reader = cmd.ExecuteReader();//执行 SQL 查询返回 SqlDataReader
while (reader.Read())                      //循环读取每一行
{
   Console.WriteLine(reader["AddressTypeID"] + " " + reader["Name"]);
                                           //输出读取的行的内容
}
reader.Close();//关闭 SqlDataReader
```

每次使用完 DataReader 对象后都应调用 Close()方法。

如果 Command 包含输出参数或返回值，那么在 DataReader 关闭之前，将无法访问这些输出参数或返回值。

注意：当 DataReader 打开时，该 DataReader 将以独占方式使用 Connection。在原 DataReader 关闭之前，将无法对 Connection 执行任何命令（包括创建另一个 DataReader）。

如果返回的是多个结果集，DataReader 会提供 NextResult()方法按顺序循环访问这些结果集。例如一次查询了两个表 AddressType 和 ContactType，使用一个 DataReader 读取这两个表的结果并输出的程序如代码 11.12 所示。

代码 11.12　使用 SqlDataReader 读取两个结果集

```
string sql = "select * from Person.AddressType;select * from Person.
ContactType";
//一个 SQL 语句中定义了两个查询
SqlCommand cmd = new SqlCommand(sql, conn);
SqlDataReader reader = cmd.ExecuteReader();//执行 SQL 查询
while (reader.Read())                            //循环读取第一个结果集
{
    Console.WriteLine(reader["AddressTypeID"] + " " + reader["Name"]);
                                                 //输出读取的内容
}
Console.WriteLine("接下来读取 ContactType 数据：");
if (reader.NextResult())                         //读取第二个结果集
{
    while (reader.Read())                        //循环读取第二个结果集
    {
        Console.WriteLine(reader["ContactTypeID"] + " " + reader["Name"]);
                                                 //输出第二个结果集
    }
}
reader.Close();
```

当 DataReader 打开时，可以使用 GetSchemaTable()方法检索有关当前结果集的架构信息。GetSchemaTable()方法将返回一个填充了行和列的 DataTable 对象，这些行和列包含当前结果集的架构信息。对于结果集的每一列，DataTable 都包含一行。架构表行的每一列都映射在结果集中返回的列属性中，其中 ColumnName 是属性的名称，而列的值为属性的值。例如读取 AddressType 表中的数据，获得该表的架构信息的程序如代码 11.13 所示。

代码 11.13　通过 SqlDataReader 获取架构信息

```
string sql = "select * from Person.AddressType";//定义一个查询
SqlCommand cmd = new SqlCommand(sql, conn);
SqlDataReader reader = cmd.ExecuteReader();//执行 SQL 查询返回 SqlDataReader
DataTable schemaTable = reader.GetSchemaTable();//获得返回的架构信息
foreach (DataRow row in schemaTable.Rows)        //循环读取每一行
{
    foreach (DataColumn column in schemaTable.Columns)//循环读取每一列
        Console.WriteLine(column.ColumnName + " = " + row[column]);
                                                 //输出列名和内容
    Console.WriteLine();
}
reader.Close();//关闭 SqlDataReader
```

11.4.2　使用 SqlDataReader 获得对象

在面向对象编程中，无论是业务层还是 UI 层操作的都是对象，这个时候希望数据层读取数据库后返回的是一个实体对象，而不是一个 SqlDataReader 数据流，这时就需要将 SqlDataReader 转换为实体对象。

使用 SqlDataReader 读取的数据一般分为两种，一种是数据库中最多找到一行数据读取出一个对应的实体实例；另外一种是数据库中返回了 0 到多行数据，读取出的就是一个实体实例的集合。例如要获得 AddressType 对象，在获得了 SqlDataReader 后，需要将其转换为对象的函数如代码 11.14 所示。

代码 11.14　通过 DataReader 获得一个实体实例的函数

```
private AddressType GetAddressTypeByReader(IDataReader reader)//传入 DataReader
{
    AddressType addressType = new AddressType();//声明对象
    //接下来读取 DataReader 为对象的每个属性赋值
    addressType.AddressTypeID = (int)reader["AddressTypeID"];
    addressType.Name = reader["Name"].ToString();
    addressType.rowguid = reader["rowguid"].ToString();
    addressType.ModifiedDate = (DateTime)reader["ModifiedDate"];
    return addressType;                          //返回对象
}
```

注意：如果某个字段在数据库中允许为空则需要判断后再取值，例如，这里允许 Name 为空的话就需要使用 Convert.IsDBNull(reader["Name"])判断当前值是否为空，如果不为空才赋值。

在创建了转换函数后，无论是返回单个实例还是返回实例集合都可以使用该函数。例如，通过 AddressTypeID 获得 AddressType 对象的程序如代码 11.15 所示。

代码 11.15　获得单个对象实例

```
string sql = "select * from Person.AddressType where AddressTypeID="+id;
//读取一个对象
SqlCommand cmd=new SqlCommand(sql,conn);
SqlDataReader reader = cmd.ExecuteReader();//执行 SQL 语句返回 SqlDataReader
if (reader.Read()) //判断是否返回了数据
{
   AddressType addressType= GetAddressTypeByReader(reader);//前面定义的函数
   reader.Close();//关闭 SqlDataReader
}
```

如果要获得当前数据库中所有的 AddressType，则对应的程序如代码 11.16 所示。

代码 11.16　获得对象实例集合

```
string sql = "select * from Person.AddressType";//定义 SQL 查询语句
SqlCommand cmd = new SqlCommand(sql, conn);
SqlDataReader reader = cmd.ExecuteReader();     //执行 SQL 查询
List<AddressType> addressTypes = new List<AddressType>();//定义一个集合
```

```
while (reader.Read())                              //循环读取每一行数据
{
   addressTypes.Add(GetAddressTypeByReader(reader));//将单个实例加入到集合中
}
reader.Close();
```

11.5　使用 DataSet 填充 SqlDataAdapter

DataSet 是数据的一种内存驻留表示形式，无论它包含的数据来自什么数据源，都会提供一致的关系编程模型。DataAdapter 用于从数据源检索数据并填充 DataSet 中的表。DataSet 表示整个数据集，其中包含对数据进行包含、排序和约束的表及表间的关系。

使用 DataSet 的方法有若干种，这些方法可以单独应用，也可以结合应用。可以：

- 以编程方式在 DataSet 中创建 DataTable、DataRelation 和 Constraint，并使用数据填充表。
- 通过 DataAdapter 用现有关系数据源中的数据表填充 DataSet。
- 使用 XML 加载和保持 DataSet 内容。

本节主要介绍 DataSet 的使用及使用 DataAdapter 填充 DataSet。

11.5.1　SqlDataAdapter 的使用

DataAdapter 用于从数据源检索数据并填充 DataSet 中的表。DataAdapter 还将对 DataSet 的更改解析回数据源。DataAdapter 使用.NET Framework 数据提供程序的 Connection 对象连接到数据源，并使用 Command 对象从数据源检索数据并将更改解析回数据源。

DataAdapter 是专门为处理脱机数据而设计的。其中最重要的一个方法就是 Fill 方法，使用这个方法可以一次性将数据库执行的结果填充到 DataSet 中，以后 DataSet 与数据库之间就保持了脱机状态。例如使用 SqlDataAdapter 将查询的所有 AddressType 数据，填充到一个 DataSet 的程序如代码 11.17 所示。

代码 11.17　使用 SqlDataAdapter 填充 DataSet

```
string sql = "select * from Person.AddressType";   //定义 SQL 查询语句
SqlCommand cmd = new SqlCommand(sql, conn);
SqlDataAdapter adapter = new SqlDataAdapter(cmd);  //声明 SqlDataAdapter
DataSet ds=new DataSet();
adapter.Fill(ds);                                  //填充一个 DataSet
```

DataAdapter 是一个双向的桥梁，既可以将数据库中的数据填充到 DataSet 中，也可以将对 DataSet 的更改写入到数据库中。对 DataSet 对象中数据的增删改对应着在 DataAdapter 中的 InsertCommand、DeleteCommand 和 UpdateCommand 属性。

在 DataAdapter 拥有对应的 Command 属性后，需要将 DataSet 中的更改写入到数据库中，只需要执行 Update()方法即可。例如在将 AddressType 填充到 DataSet 中后，更改了 DataSet 中的一个 ModifiedDate 属性，然后将 DataSet 的更改写入到数据库的程序如代码 11.18 所示。

代码 11.18　使用 SqlDataAdapter 将 DataSet 更改写入数据库

```
string sql = "select * from Person.AddressType";       //定义 SQL 语句
SqlCommand cmd = new SqlCommand(sql, conn);
```

```
SqlDataAdapter adapter = new SqlDataAdapter(cmd);//声明 SqlDataAdapter 对象
DataSet ds=new DataSet();
adapter.Fill(ds);                                    //填充数据
ds.Tables[0].Rows[0]["ModifiedDate"]=DateTime.Now;   //修改 DataSet 数据
//以下定义更新数据库的方法
SqlCommand  updateCmd=new  SqlCommand("update  Person.AddressType  set
ModifiedDate= @mdate where AddressTypeID=@id",conn);  //定义更新语句
updateCmd.Parameters.Add("@mdate",SqlDbType.DateTime,8,"ModifiedDate");
//更新参数
updateCmd.Parameters.Add("@id",SqlDbType.Int,4,"AddressTypeID");
                                                     //更新参数
adapter.UpdateCommand = updateCmd;//将更新语句的 SqlCommand 对象应用到
                         SqlDataAdapter
adapter.Update(ds);                                  //执行更新操作
```

11.5.2　DataSet 的结构

DataSet 对象的核心是数据的集合。DataSet 可以存储多个查询结果，一个结果在 DataSet 中使用 DataTable 来表示。查询结果中的数据按照行和列的形式分开，DataTable 中使用 DataRow 对象和 DataColumn 对象来表示。一行数据对应一个 DataRow，一列数据对应一个 DataColumn。DataSet 对象的结构如图 11.3 所示。

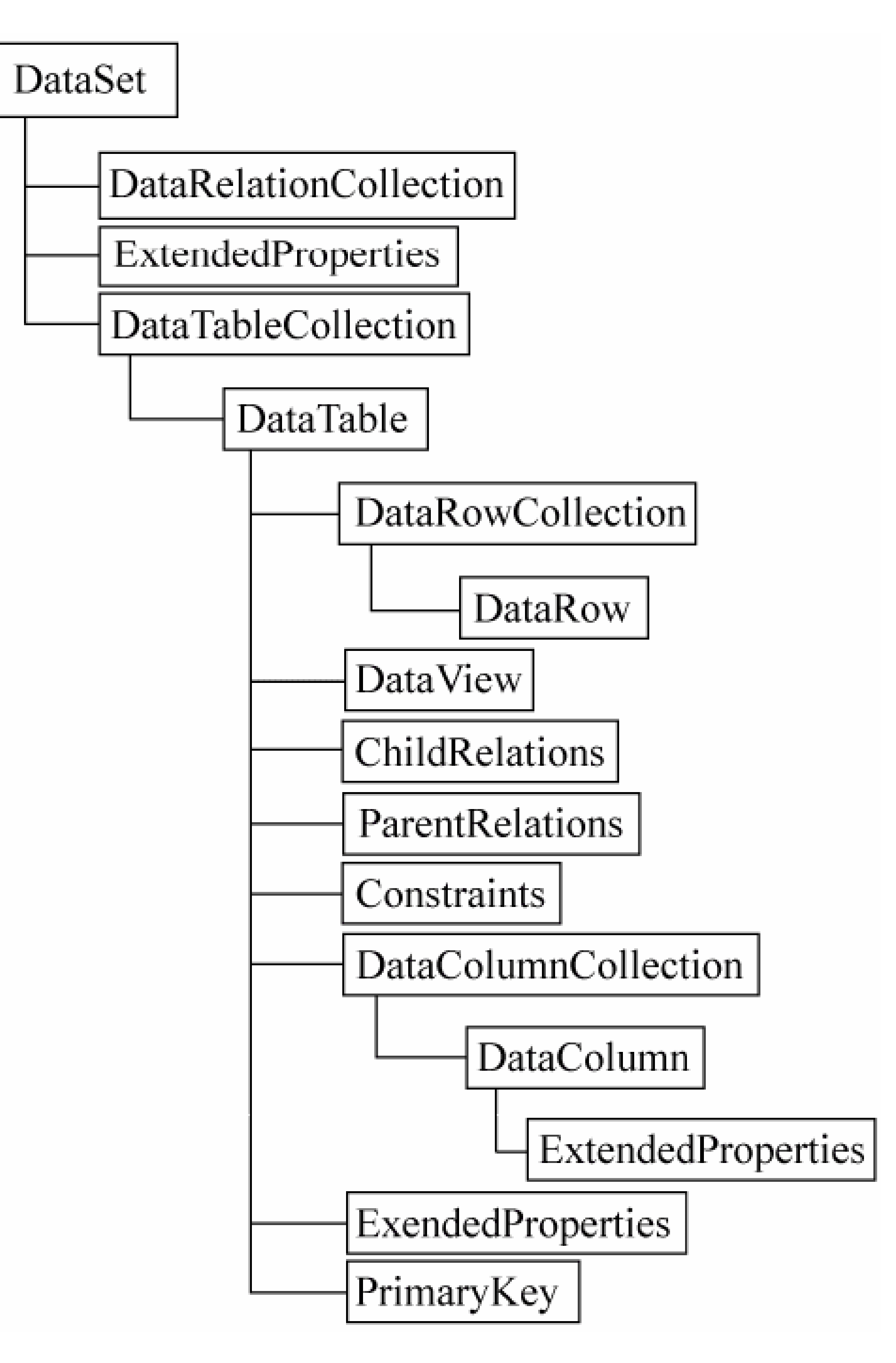


图 11.3　DataSet 的结构

11.5.3　DataSet 中的集合——DataTable

DataTable 对象在 DataSet 中用 Tables 属性表示其集合。一个查询命令中返回多个查询表结果，则对应着多个 DataTable。例如代码 11.19 所示，由于返回了两个表，所以将从 DataSet 中获得两个 DataTable。

代码 11.19　获得多个 DataTable 对象

```
string sql = "select * from Person.AddressType;select * from Person.
ContactType";
                                           //一个 SQL 语句中有多个查询
SqlCommand cmd = new SqlCommand(sql, conn);//声明 SqlCommand 对象
SqlDataAdapter adapter = new SqlDataAdapter(cmd);//声明 SqlDataAdapter 对象
DataSet ds=new DataSet();
adapter.Fill(ds);                          //填充数据
Console.WriteLine(ds.Tables.Count);        //返回 DataTable 的个数，结果为 2
```

DataSet 对象允许动态地向其中添加、删除和修改 DataTable。例如代码 11.20 所示，创建了两个 DataTable 并添加到 DataSet 中，通过 DataTable 的 TableName 可以从 DataSet 中检索到具体的 DataTable。

代码 11.20　动态添加 DataTable

```
DataSet ds = new DataSet();
string sql = "select * from Person.AddressType;";//定义 SQL 语句
SqlCommand cmd = new SqlCommand(sql, conn);        //声明 SqlCommand 对象
SqlDataAdapter adapter = new SqlDataAdapter(cmd);//声明一个 SqlDataAdapter
                                                   对象
DataTable  dt1=new DataTable();
adapter.Fill(dt1);                                 //填充数据
dt1.TableName="AddressType";                       //定义 DataTable 的名字
ds.Tables.Add(dt1);                         //向 DataSet 中添加一个 DataTable
Console.WriteLine(ds.Tables.Count);                //输出：1
sql = "select * from Person.ContactType";          //定义另一个 SQL 查询
cmd = new SqlCommand(sql, conn);
adapter = new SqlDataAdapter(cmd);
DataTable dt2 = new DataTable();
adapter.Fill(dt2);                                 //填充数据
dt1.TableName = "ContactType";
ds.Tables.Add(dt2);                                //再添加一个 DataTable
Console.WriteLine(ds.Tables.Count);                //输出：2
Console.WriteLine(ds.Tables["ContactType"].TableName);//通过 TableName 检
                                                        索 DataTable
```

11.5.4　DataSet 中的数据行——DataRow

DataRow 类表示表中包含的实际数据。DataRow 及其属性和方法用于检索、计算和处理表中的数据。在访问和更改行中的数据时，DataRow 对象会维护其当前状态和原始状态。

DataRow 以集合的形式存在于 DataTable 的 Rows 属性中，可以通过索引的方式获得一个 DataRow 实例。例如现在有 DataSet 的一个实例 ds，要获得其中第一个表的第一个 DataRow 实例对应的程序为：

```
DataRow dr = ds.Tables[0].Rows[0];
```

DataRow 实例本身提供了索引功能，以便获得一行数据中具体每列中的数据。例如要获得第一行第一列的数据，则对应程序为：

```
object obj1 = dr[0];
```

要获得第一行 Name 列的数据对应程序为：

```
object obj2 = dr["Name"];
```

获得的对象都是 object 对象，再对照数据库中定义的每一列的数据类型，可以将 object 类型转换为对应的数据类型。

使用循环语句可以检索出 DataTable 中的所有 DataRow 行。例如要输出 AddressType 中的所有 Name 数据，则对应的程序如代码 11.21 所示。

代码 11.21　获得所有的 DataRow

```
string sql = "select * from Person.AddressType;";//定义 SQL 查询语句
SqlCommand cmd = new SqlCommand(sql, conn);        //声明 SqlCommand 对象
```

```
SqlDataAdapter adapter = new SqlDataAdapter(cmd);  //声明 SqlDataAdapter 对象
DataSet ds = new DataSet();                         //定义一个 DataSet
adapter.Fill(ds);                                   //填充数据
foreach (DataRow dr in ds.Tables[0].Rows)           //循环从 DataSet 的第一个
                                                      DataTable 中读取每一行
{
    Console.WriteLine(dr["Name"]);                  //输出 Name 列的内容
}
```

11.5.5　DataSet 中的数据列——DataColumn

DataColumn 用于表示 DataTable 中列的架构。DataColumn 对象的实例下提供了 DataType 属性，该属性确定 DataColumn 所包含的数据种类。AllowDBNull、Unique 和 ReadOnly 等属性对数据的输入和更新施加限制，从而有助于确保数据完整性。

通过 DataTable 的 Columns 属性可以获得 DataColumn 对象实例的集合，通过索引的方式可以获得具体的 DataColumn 实例。例如要获得第一列的 DataColumn 实例，对应程序为：

```
DataColumn dc = ds.Tables[0].Columns[0];
```

若要获得 Name 列的 DataColumn 实例，对应的程序为：

```
DataColumn dc = ds.Tables[0].Columns["Name"];
```

同样以 AddressType 为例，要获得该表的所有列名和数据类型，则对应的程序如代码 11.22 所示。

代码 11.22　通过 DataColumn 获得列名和数据类型

```
string sql = "select * from Person.AddressType;";//定义 SQL 查询语句
SqlCommand cmd = new SqlCommand(sql, conn);         //声明 SqlCommand 对象
SqlDataAdapter adapter = new SqlDataAdapter(cmd);   //声明 SqlDataAdapter 对象
DataSet ds = new DataSet();                          //定义 DataSet
adapter.Fill(ds);                                    //填充数据
foreach (DataColumn dc in ds.Tables[0].Columns)     //循环 DataSet 中第一个
                                                      DataTable 的每一列
{
    Console.WriteLine(dc.ColumnName + " " + dc.DataType);//输出列名和数据类型
}
```

11.5.6　DataSet 中的数据视图——DataView

DataView 能够创建 DataTable 中所存储数据的不同视图，这种功能通常用于数据绑定应用程序。使用 DataView，可以使用不同排序顺序显示表中的数据，并且可以按行状态或基于筛选器表达式来筛选数据。

DataView 提供基础 DataTable 中数据的动态视图：内容、排序和成员关系会实时反映其更改。此行为不同于 DataTable 的 Select()方法，后者从表中按特定的筛选器和（或）排序顺序返回 DataRow 数组，虽然其内容反映对基础表的更改，但其成员关系和排序却仍然保持静态。DataView 的动态功能使其成为数据绑定应用程序的理想选择。

与数据库视图类似，DataView 提供了可向其应用不同排序和筛选条件的单个数据集的动态视图。但是与数据库视图不同的是，DataView 不能作为表来对待，无法提供联接表的视图。另外，还不能排除存在于源表中的列，也不能追加不存在于源表中的列（如计算列）。

例如对于 AddressType 表，若需要安装 Name 进行排序，然后输出表的 AddressTypeID 和 Name，则对应的程序如代码 11.23 所示。

代码 11.23　使用 DataView 排序

```
string sql = "select * from Person.AddressType;";//定义 SQL 查询语句
SqlCommand cmd = new SqlCommand(sql, conn);        //声明 SqlCommand 对象
SqlDataAdapter adapter = new SqlDataAdapter(cmd);  //声明 SqlDataAdapter 对象
DataSet ds = new DataSet();                        //定义 DataSet
adapter.Fill(ds);                                  //填充数据
DataView dv = new DataView(ds.Tables[0]);//使用 DataSet 中第一个 DataTable 生
                                           成 DataView
dv.Sort = "Name";                                //根据 Name 列进行排序
for (int i = 0; i < dv.Count; i++)               //循环读取 DataView 中的每一行
{
   Console.WriteLine(dv[i].Row["AddressTypeID"]+" "+dv[i].Row["Name"]);
                                                 //输出一行中的内容
}
```

系统输出排序后的结果为：

```
6 Archive
1 Billing
2 Home
3 Main Office
4 Primary
5 Shipping
```

除了排序外，DataView 还常用于数据的分页，例如，要获得 AddressType 表中 AddressTypeID 小于 5 而且又大于 2 的行，则对应的程序如代码 11.24 所示。

代码 11.24　使用 DataView 分页

```
string sql = "select * from Person.AddressType;";//定义 SQL 查询语句
SqlCommand cmd = new SqlCommand(sql, conn);        //声明 SqlCommand 对象
SqlDataAdapter adapter = new SqlDataAdapter(cmd);  //声明 SqlDataAdapter 对象
DataSet ds = new DataSet();                        //定义一个 DataSet
adapter.Fill(ds);                                  //填充数据
DataView dv = new DataView(ds.Tables[0]); //使用 DataSet 中的第一个 DataTable
                                           生成 DataView
dv.RowFilter = "AddressTypeID<5 and AddressTypeID>2";//设置过滤条件
for (int i = 0; i < dv.Count; i++)         //循环输出 DataView 中的每一行
{
   Console.WriteLine(dv[i].Row["AddressTypeID"]+" "+dv[i].Row["Name"]);
                                           //输出一行中的内容
}
```

系统返回筛选后的结果为：

```
3 Main Office
4 Primary
```

11.6 事 务 处 理

事务是合并成逻辑工作单位的操作组。它们用于控制和维护事务中每个操作的一致性和整体性，而不管系统中可能发生的错误。在一个事务中要么所有的数据操作全部成功，要么所有的操作全部失败，避免了部分操作成功部分操作失败造成的数据不一致。

11.6.1　使用 SqlTransaction 处理事务

ADO.NET 提供了 SqlTransaction 对象用于事务的处理。使用 SqlTransaction 进行事务处理操作主要有以下几步。

（1）使用 SqlConnection 实例提供的 BeginTransaction()方法，可以创建一个 SqlTransaction 实例。

（2）创建 SqlCommand 实例时将前面创建的 SqlTransaction 实例添加到构造函数中。

（3）所有 SqlCommand 命令执行完成后，执行 SqlTransaction 实例的 Commit()方法提交事务。

（4）如果在执行 SqlCommand 中发生了异常，使用 Try Catch 语句块捕捉异常，并在 Catch 语句中执行 SqlTransaction 实例的 Rollback()方法回滚整个事务。

例如一个个人财务系统，其中个人现金用 t1 表表示，个人存款用 t2 表表示。在执行取款或存款操作时需要对两个表进行更新操作，t1 增加的金额就是 t2 减少的金额。但是需要在表中添加约束，金额不能为负数。假设用户 1 有现金 50 元，银行存款 50 元，则对应的创建数据库表和数据的脚本如代码 11.25 所示。

代码 11.25　创建基础表

```
CREATE TABLE t1 --创建一个测试表 t1
(
    c1 INT PRIMARY KEY,
    c2 INT NOT NULL,
    CONSTRAINT CK t1c2 CHECK(c2>=0) --CHECK 约束
)
GO
CREATE TABLE t2 --创建一个测试表 t2
(
    c1 INT PRIMARY KEY,
    c2 INT NOT NULL,
    CONSTRAINT CK t2c2 CHECK(c2>=0) --CHECK 约束
)
GO
--接下来插入示例数据
INSERT INTO t1 VALUES(1,50);
INSERT INTO t2 VALUES(1,50);
```

现在使用 SqlTransaction 对象进行事务处理，需要从银行取款 30 元，对应的程序如代码 11.26 所示。

代码 11.26　使用 SqlTransaction 进行事务处理

```
using (SqlConnection conn = new SqlConnection(sqlconn))//声明一个连接
{
    conn.Open();                                        //打开连接
    SqlTransaction trans = conn.BeginTransaction();//使用 SqlConnection 创
                                                          建事务
    try
    {
        string sql = "update t1 set c2=c2+30 where c1=1";//取款现金增加 30 元
        SqlCommand cmd = new SqlCommand(sql, conn, trans);
        cmd.ExecuteNonQuery();                          //执行现金增加 30 元的操作
        string sql2 = "update t2 set c2=c2-30 where c1=1";//取款时存款减少
                                                              30 元
        SqlCommand cmd2 = new SqlCommand(sql2, conn, trans);
        cmd2.ExecuteNonQuery();                         //执行存款减少 30 元的操作
        trans.Commit();                                 //提交事务
        Console.WriteLine("执行成功");
    }
    catch (Exception ex)
    {
        trans.Rollback();                               //发生了异常，回滚事务
        Console.WriteLine("执行失败，因为：" + ex.Message);
    }
}
```

第一次执行代码 11.26 后再查询数据库，可以看到 t1 表中的金额变成了 80 元，而 t2 表中的金额减少为 20 元。再次执行以上代码，以再次取款 30 元，但是由于 t2 表中的金额只有 20 元，取款 30 元将会造成银行存款为负数，违反 t2 表的 CHECK 约束。所以在 cmd2.ExecuteNonQuery();时必将发生异常，系统捕捉到异常后将执行事务的回滚操作，将已经执行的 cmd 语句回滚。查询数据库中可以看到，t1 表仍然是 80 元，t2 表仍然是 20 元。

11.6.2　使用 TransactionScope 处理分布式事务

TransactionScope 类通过隐式在分布式事务中登记连接，使代码块事务化。必须在 TransactionScope 块的结尾调用 Complete()方法，然后再离开该代码块。离开代码块将调用 Dispose()方法。如果引发的异常造成代码离开范围，将认为事务已中止。

一般采用 using 代码块，以确保在退出 using 代码块时，在 TransactionScope 对象上调用 Dispose()。如果无法提交或回滚挂起的事务，可能会对性能造成严重影响，因为 TransactionScope 的默认超时为 1 分钟。如果不使用 using 语句，必须在 Try 代码块中执行所有工作，并在 Finally 代码块中显式调用 Dispose()方法。

如果在 TransactionScope 中发生异常，事务将标记为不一致并被弃用。在 TransactionScope 断开后，事务将回滚。如果未发生任何异常，参与的事务将提交。

TransactionScope 并不像 SqlTransaction 一样面向某个数据库连接，在 TransactionScope 中可以使用不同的数据库连接操作分布式事务。如果 TransactionScope 中使用的几个连接指向同一个数据库实例，则其中的事务处理与 SqlTransaction 相同。

🔔注意：在不同的数据库实例连接中使用 TransactionScope 进行分布式事务处理时，必须要为服务器启动分布式处理事务的服务 Distributed Transaction Coordinator（简称 DTC）。

同样以前面提到的 t1 表和 t2 表的操作为例，使用 TransactionScope 执行事务操作的程序如代码 11.27 所示。

代码 11.27　使用 TransactionScope 进行事务处理

```
try
{
    using (TransactionScope trans = new TransactionScope())
                                        //使用 TransactionScope 启动事务
    {
        using (SqlConnection conn = new SqlConnection(sqlconn))
                                        //在一个连接中执行了操作
        {
            conn.Open();                //打开连接
            string sql = "update t1 set c2=c2+30 where c1=1";
            SqlCommand cmd = new SqlCommand(sql, conn);
            cmd.ExecuteNonQuery();      //执行现金增加 30 元的操作
        }
        using (SqlConnection conn = new SqlConnection(sqlconn))
                                        //在另外一个连接中执行操作
        {
            conn.Open();                //再打开一个连接
            string sql2 = "update t2 set c2=c2-30 where c1=1";
            SqlCommand cmd2 = new SqlCommand(sql2, conn);
            cmd2.ExecuteNonQuery();     //执行存款减少的操作
        }
        trans.Complete();               //提交事务
        Console.WriteLine("操作成功");
    }
}
catch (Exception ex)                    //捕捉到异常
{
    Console.WriteLine(ex.Message);      //输出异常消息内容
}
```

代码中使用 using 语句的 TransactionScope 执行事务操作，在发生异常时不需要像 SqlTransaction 那样执行 RollBack 操作。因为在异常发生时系统没有执行 TransactionScope 实例下的 Complete()方法，离开 using 块时执行了自动执行 Dispose()方法，所以 using 块中的事务全部被回滚了。

🔔注意：使用 TransactionScope 对象时，需要添加对 System.Transactions 命名空间的引用，该命名空间在 System.Transactions.dll 程序集中。

11.7　使用数据访问应用程序块

前面介绍了使用 SqlConnection 建立数据库连接、使用 SqlCommand 执行数据库操作、

使用 SqlDataReader 读取数据流、使用 SqlDataAdapter 填充 DataSet 对象，进行一次数据库操作就要涉及这么多对象。在实际项目中对数据库的操作少则几百，多则成千上万，那就要反复地编写程序调用这些对象。为了减少这种重复的编码工作，微软提供了一个程序集 Data Access Application Block，帮助程序员使用 ADO.NET 访问数据库。

11.7.1　数据访问应用程序块简介

数据访问应用程序块（Data Access Application Block）是一个.NET 组件，它包含经过优化的数据访问代码，可以帮助用户调用存储过程，以及向 SQL Server 数据库发出 SQL 文本命令。它返回 SqlDataReader、DataSet 和 XmlReader 对象。可以在自己的基于.NET 的应用程序中将其作为构造块来使用，以减少需要创建、测试和维护的自定义代码的数量。读者可以到微软的官方网站下载 Data Access Application Block 的程序集和源代码。

Data Access Application Block 在 ADO.NET 1.1 时便得到了广泛使用，它将有关访问 Microsoft SQL Server 数据库的性能和资源管理方面的最佳做法封装在一起。用户可以很方便地在自己的基于.NET 的应用程序中将其作为构造块使用，从而减少需要创建、测试和维护的自定义代码的数量。

使用 Data Access Application Block 可以：

- 调用存储过程或 SQL 文本命令。
- 指定参数详细信息。
- 返回 SqlDataReader、DataSet 或 XmlReader 对象。
- 使用强类型的 DataSet。

Data Access Application Block 的核心是 SqlHelper 类和 SqlHelperParameterCache 类。

- SqlHelper 类提供了一组静态方法，可以用来对 SQL Server 数据库执行多种不同类型的命令。
- SqlHelperParameterCache 类提供命令参数缓存功能，可以用来提高性能。该类由许多 Execute()方法（尤其是那些只执行存储过程的重载方法）在内部使用。数据访问客户端也可以直接使用它来缓存特定命令的特定参数集。

Data Access Application Block 的主要元素和主要方法如图 11.4 所示。

SqlHelper 类用于通过一组静态方法封装数据访问功能。该类不能被继承或实例化，因此将其声明为包含私有构造函数的不可继承类。

在 SqlHelper 类中实现的每种方法都提供了一组一致的重载。它提供了一种很好的使用 SqlHelper 类来执行命令的模式，同时为开发人员选择访问数据的方式提供了必要的灵活性。每种方法的重载都支持不同的方法参数，因此开发人员可以确定传递连接、事务和参数信息的方式。在 SqlHelper 类中实现的方法包括：

- ExecuteNonQuery()：此方法用于执行不返回任何行或值的命令。这些命令通常用于执行数据库更新，但也可用于返回存储过程的输出参数。
- ExecuteReader()：此方法用于返回 SqlDataReader 对象，该对象包含由某一命令返回的结果集。
- ExecuteDataset()：此方法返回 DataSet 对象，该对象包含由某个命令返回的结果集。
- ExecuteScalar()：此方法返回一个值。该值始终是该命令返回的第一行的第一列。

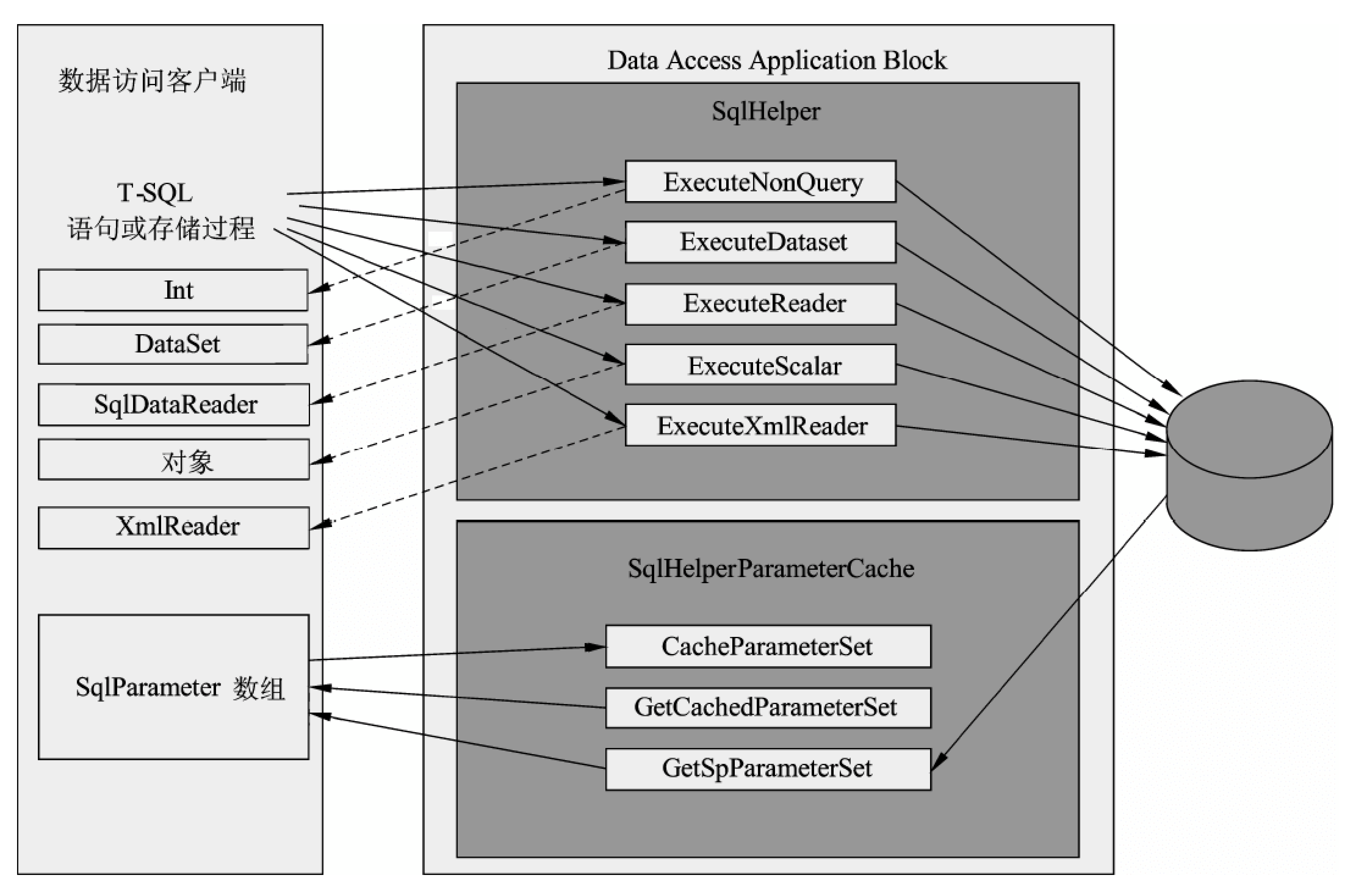


图 11.4　Data Access Application Block 的主要元素和方法

- ExecuteXmlReader()：此方法返回 FORXML 查询的 XML 片段。
- FillDataset()：此方法类似于 ExecuteDataset，不同之处在于可以传入预先存在的 DataSet，从而允许添加附加表。
- UpdateDataset()：此方法使用现有的连接和用户指定的更新命令来更新 DataSet。它通常与 CreateCommand 命令结合使用。
- CreateCommand()：此方法允许提供存储过程和可选参数，从而简化了 SQL 命令对象的创建过程。此方法通常与 UpdateDataset 结合使用。
- ExecuteNonQueryTypedParams()：此方法使用数据行而不是参数来执行非查询操作。
- ExecuteDatasetTypedParams()：此方法使用数据行而不是参数来执行 DataSet 创建操作。
- ExecuteReaderTypedParams()：此方法使用数据行而不是参数来返回数据读取器。
- ExecuteScalarTypedParams()：此方法使用数据行而不是参数来返回标量。
- ExecuteXmlReaderTypedParams()：此方法使用数据行而不是参数来执行 XmlReader。

SqlHelperParameterCache 类提供了 3 个可以用来管理参数的公共共享方法，它们是：

- CacheParameterSet()：用于将 SqlParameters 数组存储到缓存中。
- GetCachedParameterSet()：用于检索缓存参数数组的副本。
- GetSpParameterSet()：一种重载方法，用于检索指定存储过程的相应参数（首先查询一次数据库，然后缓存结果以便将来查询）。

11.7.2　数据访问应用程序块的使用

若要使用数据访问应用程序块，必须添加对程序集 Microsoft.ApplicationBlocks.Data.dll

的引用，然后添加命名空间 Microsoft.ApplicationBlocks.Data，大部分功能都是通过 SqlHelper 类的静态方法来实现的。例如要获得 AdventureWorks2012 数据库中 Person.AddressType 表的所有数据对应的 DataSet 对象，则使用 SqlHelper 调用的程序如代码 11.28 所示。

代码 11.28　使用 SqlHelper 获得 DataSet

```
string sqlconn = "server=IBM-PC;database=AdventureWorks2012;uid=sa;
pwd=p@ssw0rd";
DataSet ds = SqlHelper.ExecuteDataset(sqlconn, //数据库连接
CommandType.Text,                              //命令类型为 T-SQL 语句
"select * from Person.AddressType");           //具体的 T-SQL 语句内容
```

对比前面使用 SqlConnection、SqlCommand、SqlDataReader 和 SqlDataAdapter 这几个对象来获得一个 DataSet 的代码，使用 SqlHelper 编写代码简单了很多。

除了简化返回 DataSet 的方法外，SqlHelper 还可以很简单地使用参数。SqlHelper 中提供了 params object[] parameterValues 类型的参数，也就是说，可以不声明 SqlParameter 对象而直接将参数值依次传入方法中。例如，AdventureWorks 数据库中提供了存储过程 GetDeparmentByGroupName，该存储过程需要参数@groupName，若要查询研发部分则对应的程序如代码 11.29 所示。

代码 11.29　在 SqlHelper 中使用参数

```
string sqlconn = "server=IBM-PC;database=AdventureWorks2012;uid=sa;
pwd=p@ssw0rd";
DataSet ds = SqlHelper.ExecuteDataset(sqlconn, //数据库连接
"GetDeparmentByGroupName",                     //要执行的存储过程的名字
"Research and Development");                   //传入的参数内容
```

这里是传入了一个参数，如果需要传入多个参数，则只需在该方法中继续按照存储过程中参数定义的顺序添加即可。

注意： *如果是在 SQL 语句中使用参数，则必须使用 SqlParameter 声明参数。因为在存储过程中定义了参数的顺序和数据类型，而 SQL 语句中却没有，所以才要求使用 SqlParameter 来指定参数值对应的参数。*

其他方法返回一个 object 对象、返回受影响的行数、返回 XmlReader 等操作和返回 DataSet 的方法的使用相似，这里不再举例。

使用 SqlHelperParameterCache 类可以管理参数。通过使用 CacheParameterSet()方法，可以缓存 SqlParameter 对象数组。此方法通过将连接字符串和命令文本连接起来创建一个键，然后将参数数组存储在 Hashtable 中。

为了从缓存中检索参数，使用了 GetCachedParameterSet()方法。此方法返回一个 SqlParameter 对象数组，这些对象已使用特定缓存（该缓存与传递给该方法的连接字符串和命令文本相对应）中参数的名称、方向和数据类型等进行了初始化。

注意： *用做参数集的键的连接字符串通过简单的字符串比较进行匹配。用于从 GetCachedParameterSet()中检索参数的连接字符串必须与通过 CacheParameterSet()来存储这*

些参数的连接字符串完全相同。语法不同的连接字符串即使语义相同，也不会被认为是匹配的。

例如，要使用 SQL 语句中带参数的形式来查询数据库，缓存参数和获得缓存中参数并查询的程序如代码 11.30 所示。

代码 11.30　使用 SqlHelperParameterCache 管理参数

```
string sqlconn = "server=IBM-PC;database=AdventureWorks2012;uid=sa;pwd=
p@ssw0rd";
string sql = "SELECT * FROM HumanResources.Employee WHERE BusinessEntityID=
@mgrID AND JobTitle=@title";//定义 SQL 查询语句
//以下缓存参数
SqlParameter[] pars = new SqlParameter[2];
pars[0] = new SqlParameter("@mgrID", SqlDbType.Int);
pars[1] = new SqlParameter("@title", SqlDbType.VarChar, 50);
SqlHelperParameterCache.CacheParameterSet(sqlconn, sql, pars);
//以下获得缓存中的参数
SqlParameter[] storedParams = new SqlParameter[2];
storedParams = SqlHelperParameterCache.GetCachedParameterSet(sqlconn, sql);
storedParams[0].Value = 3;
storedParams[1].Value = "Design Engineer";
//以下是将参数传入，执行 SQL 查询
DataSet ds = SqlHelper.ExecuteDataset(sqlconn, CommandType.Text, sql,
storedParams);
```

11.8　使用 LINQ 操作数据库

从.Net Framework 3.5 开始就引入了 LINQ（Language Integrated Query，语言集成查询）。使用 LINQ 可以实现程序中对象与关系数据库的映射；能够自动生成 SQL 语句，完成数据库操作。本节将主要介绍 LINQ 在数据库操作中的使用。

11.8.1　LINQ 基础

LINQ 是集成在.NET 编程语言中的一种特性。它已成为编程语言的一个组成部分，在编写程序时可以得到很好的编译时语法检查，丰富的元数据，智能感知、静态类型等强类型语言的好处。同时它还可以方便地对内存中的信息进行查询而不仅仅是外部数据源。

LINQ 使查询成了.NET 中头等的编程概念，被查询的数据可以是 XML（LINQ to XML）、Databases（LINQ to SQL、LINQ to Dataset、LINQ to Entities）和对象（LINQ to Objects）。LINQ 也是可扩展的，允许用户建立自定义的 LINQ 数据提供者（比如 LINQ to Amazon、LINQ to NHibernate、LINQ to LDAP）。由于此处主要是讲解使用 LINQ 来查询 SQL Server 数据库，所以这里的重点也就是 LINQ to SQL。

注意：要使用 LINQ，要求项目必须是.NET Framework 3.5 以上版的。

在 LINQ to SQL 中，关系数据库的数据模型映射到用开发人员所用的编程语言表示的对象模型。当应用程序运行时，LINQ to SQL 会将对象模型中的语言集成查询转换为 SQL，

然后将它们发送到数据库进行执行。当数据库返回结果时，LINQ to SQL 会将它们转换回可以用自己的编程语言处理的对象。

LINQ 支持 SQL Server 数据库、XML 文档、ADO.NET 数据集，以及支持 IEnumerable 或泛型 IEnumerable<(Of<(T>)>)接口的任意对象集合。

系统为 LINQ 查询提供了专门的查询表达式，查询表达式必须以 from 子句开头。另外，查询表达式还可以包含子查询，子查询也是以 from 子句开头。from 子句指定以下内容：

- 将对其运行查询或子查询的数据源。
- 一个本地范围变量，表示源序列中的每个元素。

例如一个考试分数的数值，由于数组支持 IEnumerable 接口，可以使用 LINQ 的查询表达式来查询这个数值，要获得并输出所有及格分数的 LINQ 查询如代码 11.31 所示。

代码 11.31　简单的 LINQ 查询

```
static void Main(string[] args)
{
    int[] marks = { 60, 77, 82, 46, 59, 98, 100, 84 };//定义一个数组
    var good = from m in marks  //使用 LINQ 找到所有数值大于等于 60 的数字
               where m >= 60
               select m;
    foreach (int mark in good)  //循环每一个结果
    {
        Console.WriteLine(mark);//输出结果
    }
}
```

LINQ 查询中除了使用查询表达式以外还可以使用 Lambda 表达式。所有 Lambda 表达式都使用 Lambda 运算符=>，该 Lambda 运算符的左边是输入参数（如果有），右边包含表达式或语句块。Lambda 用在基于方法的 LINQ 查询中，作为诸如 Where()和 Where(IQueryable,String,array<Object>[]()[])等标准查询运算符方法的参数。例如前面提到的查询所有及格的分数，若使用 Lambda 表达式来查询如代码 11.32 所示。

代码 11.32　使用 Lambda 表达式的 LINQ 查询

```
static void Main(string[] args)
{
    int[] marks = { 60, 77, 82, 46, 59, 98, 100, 84 };
    var good = marks.Where(m => m >= 60);//使用 Lambda 表达式查询数值大于等于
                                          60 的数字
    foreach (int mark in good)  //循环每一个结果
    {
        Console.WriteLine(mark);//输出结果
    }
}
```

LINQ 查询中更多针对的是对象集合而不是简单的数据类型的集合。对象集合查询表达式的写法相同，在查询表达式中可以直接使用对象的属性和方法。例如在一个学生的集合中包含了多个学生对象，若要查询所有性别为男并且年龄小于 25 岁的学生的 LINQ 查询如代码 11.33 所示。

代码 11.33　使用 LINQ 查询对象集合

```
class Program
{
static void Main(string[] args)
{
    List<Student> students = new List<Student>//声明 Student 对象的集合
    {
new Student{ StudentID=1, Sex=true, Name="何名", Birthday=Convert.
ToDateTime("1984-1-1")},
new Student{ StudentID=2, Sex=false, Name="宋杰", Birthday=Convert.
ToDateTime("1983-5-1")},
new Student{ StudentID=3, Sex=true, Name="刘晓", Birthday=Convert.
ToDateTime("1983-2-15")},
new Student{ StudentID=4, Sex=false, Name="章婷", Birthday=Convert.
ToDateTime("1985-3-8")}
    };
//接下来使用 LINQ 查询所有年龄小于 25 岁的男性
    var names = from s in students
              where s.Sex == true&&s.GetAge()<25    //调用对象的方法
              select s.Name;
    foreach (string name in names)                  //循环输出结果
    {
        Console.WriteLine(name);
    }
}
}
public class Student                                //定义 Student 类
{
//以下是定义类中的属性
    public string Name { get; set; }
    public int StudentID { get; set; }
    public bool Sex { get; set; }
    public DateTime Birthday { get; set; }
//以下是定义类中的方法
    public int GetAge()
    {
        return DateTime.Now.Year - Birthday.Year;
    }
}
```

11.8.2　创建 LINQ to SQL

LINQ to SQL 是适合不需要映射到概念模型的开发人员使用的有用工具。通过使用 LINQ to SQL，可以直接在现有数据库架构上直接使用 LINQ 编程模型。LINQ to SQL 使开发人员能够生成表示数据的.NET Framework 类。这些生成的类直接映射到数据库表、视图、存储过程和用户定义的函数，而不映射到概念数据模型。

使用 LINQ to SQL 时，除了其他数据源（如 XML）外，开发人员还可以使用与内存集合和 DataSet 相同的 LINQ 编程模式，直接编写针对存储架构的代码。

可以使用 VS 2010 的对象关系设计器，它提供了开发人员用于实现许多 LINQ to SQL 功能的用户界面。在进行 LINQ to SQL 的操作之前，需要创建一个示例数据库，该数据库中包含了 Student 和 Class 两个表，并包含一些初始化数据。创建示例数据库的脚本如代码 11.34 所示。

代码 11.34　创建示例数据库

```
CREATE TABLE Class --创建班级表
(
    ClassID INT PRIMARY KEY,
    ClassName NVARCHAR(20) NOT NULL
)
GO
CREATE TABLE Student --创建学生表
(
    StudentID INT IDENTITY PRIMARY KEY,
    NAME NVARCHAR(10) NOT NULL,
    Sex BIT NOT NULL,
    BirthDay DATETIME NOT NULL,
    ClassID INT NOT NULL,
    --以下定义的是外键约束
    CONSTRAINT FK Student Class FOREIGN KEY(ClassID) REFERENCES Class
    (ClassID)
)
GO
--以下是插入的示例数据
INSERT INTO Class VALUES(1,'200801');
INSERT INTO Class VALUES(2,'200802');
INSERT INTO Student ([NAME],Sex,BirthDay,ClassID) VALUES ('张三',1,'1982-
1-12',1)
INSERT INTO Student ([NAME],Sex,BirthDay,ClassID) VALUES ('李四',1,'1983-
11-2',1)
INSERT INTO Student ([NAME],Sex,BirthDay,ClassID) VALUES ('何欢',1,'1982-
8-6',2)
INSERT INTO Student ([NAME],Sex,BirthDay,ClassID) VALUES ('晏婉',0,'1983-
6-15',2)
```

在创建好示例数据库后，接下来就是在 VS 2010 中创建 LINQ to SQL 的项目，具体操作如下。

（1）新建控制台应用程序，命名为 LINQTest，该项目必须是 NET Framework 3.5 以上版本。

（2）在解决方案资源管理器中为 LINQTest 项目添加新项 LINQ to SQL 类，并命名为 StuDataClasses，如图 11.5 所示。

（3）在 VS 2010 的服务器资源管理器中添加测试数据库的连接。

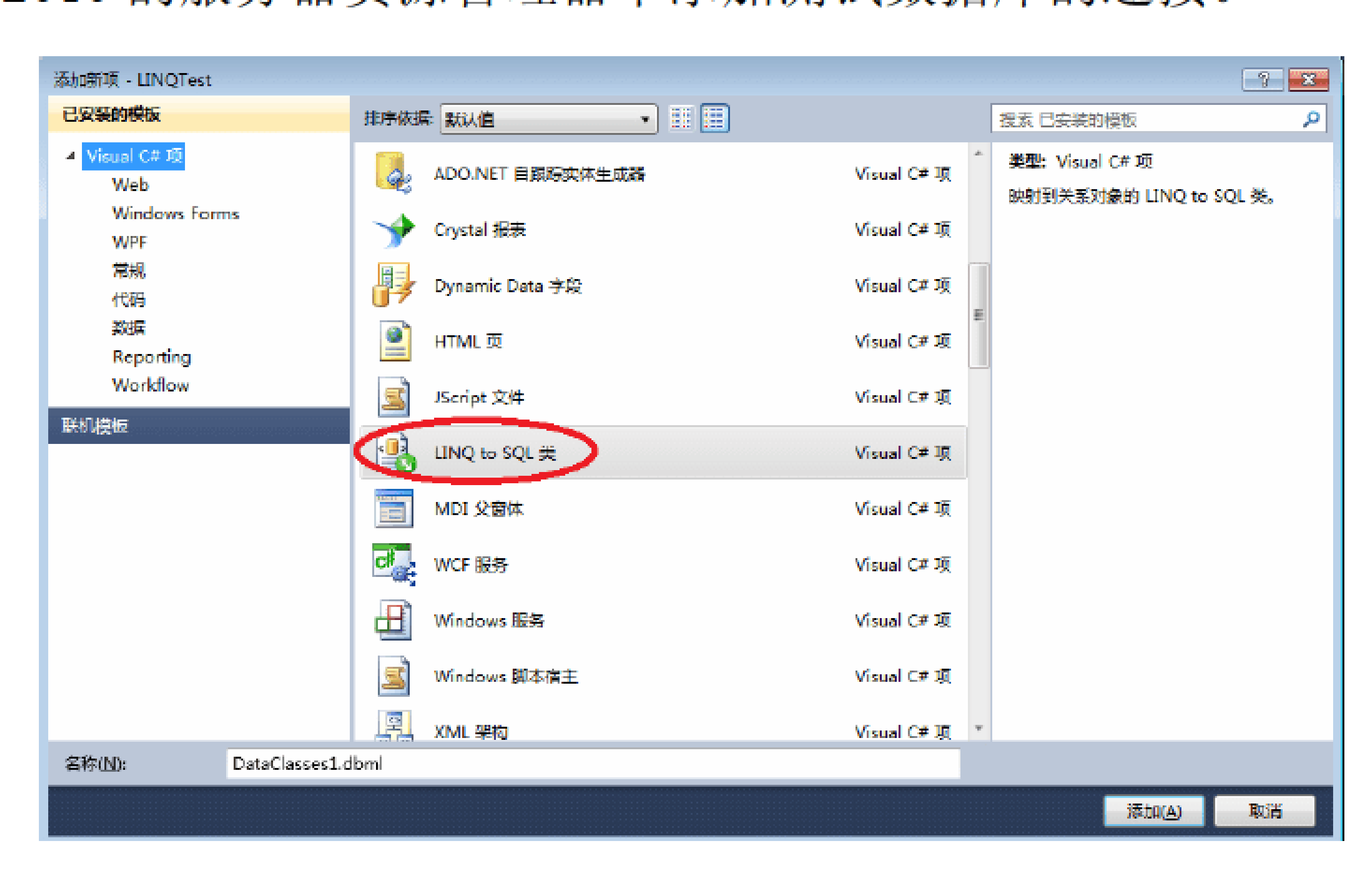


图 11.5　添加 LINQ to SQL 类

（4）将服务器资源管理器中的 Student 类和 Class 类，拖曳到 LINQ to SQL 的对象关系设计器中，如图 11.6 所示。

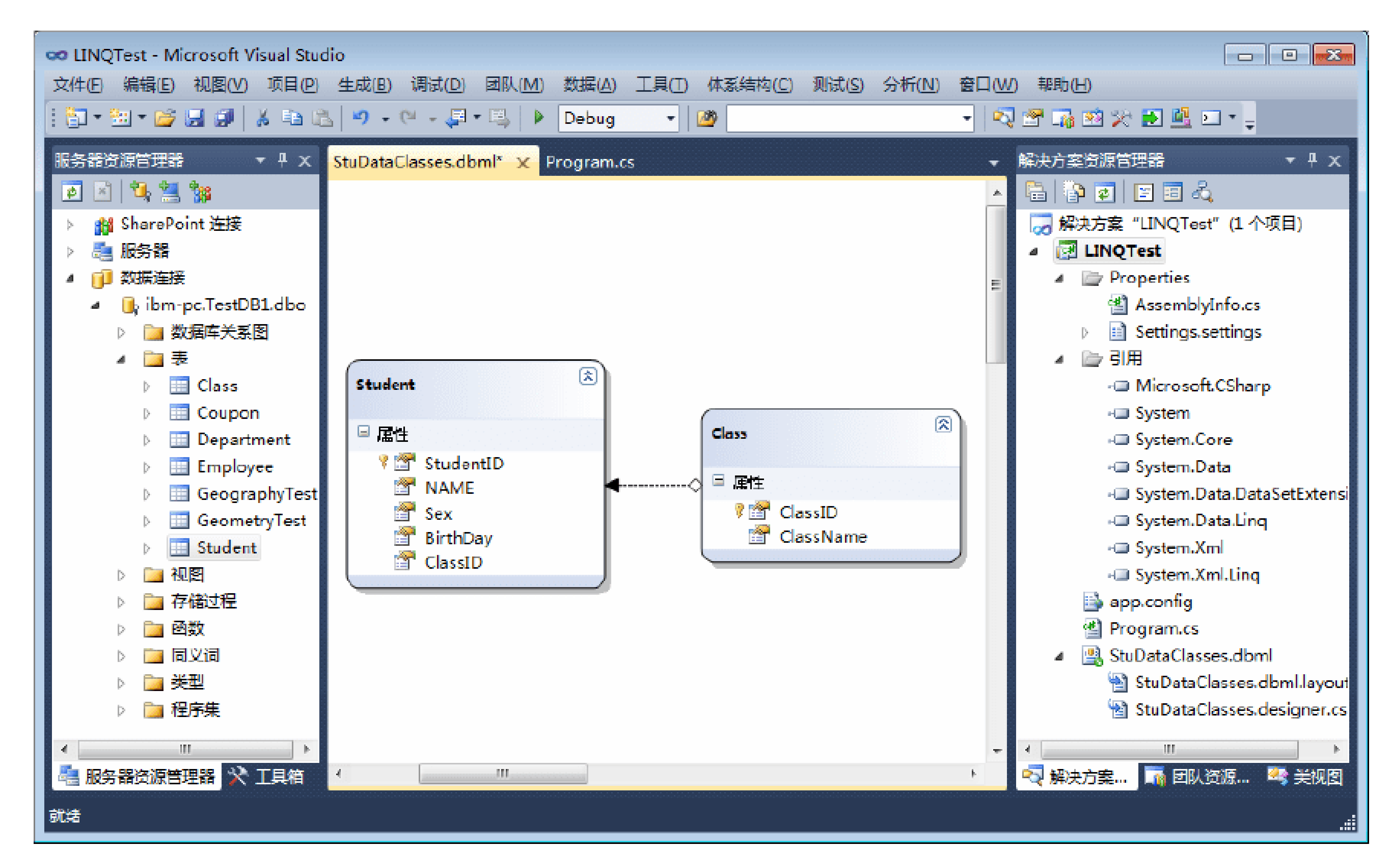


图 11.6　LINQ to SQL 设计界面

（5）保存 StuDataClasses.dbml 文件，在系统中将会建立 StuDataClassesDataContext 类，该类就是用于 LINQ 操作的类，通过该类的属性可以找到 Class 对象和 Student 对象。

（6）构造 StuDataClassesDataContext 的实例，输出其中的所有学生姓名的程序如代码 11.35 所示。

代码 11.35　通过 StuDataClassesDataContext 查询学生姓名

```
static void Main(string[] args)
{
   StuDataClassesDataContext context=new StuDataClassesDataContext();
                                                   //声明对象
   var students=from s in context.Student  //LINQ 查询所有学生的名字
                select s.NAME;
   foreach(var name in students)           //循环输出所有学生的姓名
   {
      Console.WriteLine(name);
   }
}
```

11.8.3　使用 LINQ 进行多表查询

在前面的例子中，通过建立 LINQ to SQL 类便可查询数据库中的数据。除了查询单个表的数据外，使用 LINQ 还可以进行多表的连接，然后进行查询。例如要输出学生的姓名和学生所在的班级名，则通过 LINQ 查询的程序如代码 11.36 所示。

代码 11.36　使用 LINQ 的多表连接查询

```
static void Main(string[] args)
{
```

```
    StuDataClassesDataContext context=new StuDataClassesDataContext();
    var students = from s in context.Student
                 from c in context.Class
                 where s.ClassID == c.ClassID   //相当于连接两个表
                 select new          //返回了学生的姓名和班级名
                 {
                    s.NAME,
                    c.ClassName
                 };
    foreach(var o in students)      //循环输出所有学生的姓名和对应班级名
    {
       Console.WriteLine(o.NAME+" "+o.ClassName);
    }
}
```

该 LINQ 查询生成的 SQL 语句为：

```
SELECT [t0].[NAME], [t1].[ClassName]
FROM [dbo].[Student] AS [t0], [dbo].[Class] AS [t1]
WHERE [t0].[ClassID] = [t1].[ClassID]
```

该代码相当于是内连接的 SQL 语句，如果系统进行外连接，则对应的 LINQ 查询如代码 11.37 所示。

代码 11.37　使用 LINQ 进行外连接查询

```
var students = from s in context.Student
         join c in context.Class
         on s.ClassID equals c.ClassID into cla
         from cl in cla.DefaultIfEmpty()       //这里相当于一个外连接操作
         select new
         {
            s.NAME,
            cl.ClassName
         };
```

生成的 SQL 语句为：

```
SELECT [t0].[NAME], [t1].[ClassName] AS [ClassName]
FROM [dbo].[Student] AS [t0]
LEFT OUTER JOIN [dbo].[Class] AS [t1] ON [t0].[ClassID] = [t1].[ClassID]
```

除了连接查询外，LINQ 还支持嵌套查询。例如要查询每个班的名字和班级中的学生数量，则对应的 LINQ 查询如代码 11.38 所示。

代码 11.38　使用 LINQ 进行嵌套查询

```
var students = from c in context.Class
         join s in context.Student
         on c.ClassID equals s.ClassID into st//嵌套查询
         select new//输出班级的名字和班级中的学生数
         {
           c.ClassName,
           StudentCount=st.Count()
         };
```

生成的 SQL 语句为：

```
SELECT [t0].[ClassName], (
    SELECT COUNT(*)
    FROM [dbo].[Student] AS [t1]
    WHERE [t0].[ClassID] = [t1].[ClassID]
    ) AS [StudentCount]
FROM [dbo].[Class] AS [t0]
```

这里使用到了聚合函数 Count()，其他聚合函数 Min()、Max()、Sum()等都可以在 LINQ 中使用。

11.8.4　使用 LINQ 的其他查询

除了进行单表查询、多表连接查询、嵌套查询以外，LINQ 还支持 Group By 的分组查询。例如在查询学生表中，按照学生的性别进行分组，获得男女学生各自的人数，则对应的 LINQ 查询如代码 11.39 所示。

代码 11.39　在 LINQ 中使用分组查询

```
var students = from s in context.Student
           group s by s.Sex into sex //使用 Sex 属性进行分组
           select new                //输出性别和性别对应的人数
           {
               Sex = sex.Key,
               Count = sex.Count()
           };
```

生成的对应 SQL 语句为：

```
SELECT COUNT(*) AS [Count], [t0].[Sex]
FROM [dbo].[Student] AS [t0]
GROUP BY [t0].[Sex]
```

除了支持分组外，LINQ 还支持排序。例如将学生表中的所有学生按照学生姓名逆向排序，则对应的 LINQ 查询如代码 11.40 所示。

代码 11.40　使用 LINQ 进行排序

```
var students = from s in context.Student
           orderby s.NAME descending //使用 Name 进行逆向排列
           select s.NAME;
```

生成的 SQL 查询语句为：

```
SELECT [t0].[NAME]
FROM [dbo].[Student] AS [t0]
ORDER BY [t0].[NAME] DESC
```

如果不需要逆向排序，直接将 descending 关键字去掉即可。

另外在 SQL 语句中，Union 操作也能够在 LINQ 中得以实现。例如要将查询的所有学生姓名和所有班级的名字进行联合，则对应的 LINQ 查询如代码 11.41 所示。

代码 11.41　使用 LINQ 进行 Union 操作

```
var students = (from s in context.Student
            select s.NAME).Concat    //联合下一个查询
```

```
          (from c in context.Class
           select c.ClassName);
```

系统产生的 SQL 语句为：

```
SELECT [t2].[NAME]
FROM (
   SELECT [t0].[NAME]
   FROM [dbo].[Student] AS [t0]
   UNION ALL
   SELECT [t1].[ClassName]
   FROM [dbo].[Class] AS [t1]
   ) AS [t2]
```

注意：Concat()方法生成的是 UNION ALL 操作，如果希望执行的是 UNION 操作，则需要使用 Union()方法。

使用 LINQ 不仅能够执行简单的 SQL 查询，还可以用于数据库分页。例如对学生表进行分页查询，每页 2 个数据，返回第 2 页的数据的 LINQ 查询如代码 11.42 所示。

代码 11.42　使用 LINQ 进行分页查询

```
var students = (from s in context.Student
          orderby s.NAME
          select s.NAME).Skip(2).Take(2);  //分页操作
```

系统生成的 SQL 语句为：

```
SELECT [t1].[NAME]
FROM (
   SELECT ROW_NUMBER() OVER (ORDER BY [t0].[NAME]) AS [ROW_NUMBER],
   [t0].[NAME]
   FROM [dbo].[Student] AS [t0]
   ) AS [t1]
WHERE [t1].[ROW_NUMBER] BETWEEN @p0 + 1 AND @p0 + @p1
ORDER BY [t1].[ROW_NUMBER]
```

生成的 SQL 语句中使用了 ROW_NUMBER()用于数据库的分页。该 LINQ 查询中 Skip(2)表示跳过 2 行数据，而 Take(2)相当于 TOP(2)取得结果集中的前两行数据。

SQL Server 中有些操作比如 LIKE 关键字并没有在 LINQ 中提供对应的关键字，但是可以通过使用 SqlMethods 类提供的方法来表示。例如要查询学生表中所有姓张的学生姓名，则对应的 LINQ 查询如代码 11.43 所示。

代码 11.43　使用 SqlMethods 类进行 LINQ 的 LIKE 查询

```
var students = from s in context.Student
         where SqlMethods.Like(s.NAME,"张%")       //对 Name 进行 LIKE 查询
         select s.NAME;
```

系统生成的 SQL 语句为：

```
SELECT [t0].[NAME]
FROM [dbo].[Student] AS [t0]
WHERE [t0].[NAME] LIKE @p0
```

注意：SqlMethods 在 System.Data.Linq.SqlClient 命名空间中时，需要添加对该命名空间的引用才能正常使用。

11.8.5　使用 LINQ to SQL 修改数据

前面介绍的都是对数据库的查询操作，LINQ to SQL 对象与 DataSet 类似，可以通过向其添加对象、修改对象或者删除对象，然后执行其中的 SubmitChanges()方法将对象的变更应用到数据库中。例如要添加一个新的班级，则需要先创建一个新的 Class 对象：

```
Class c=new Class{ ClassID=3, ClassName="200803"};
```

然后将该对象添加到 LINQ to SQL 的类中：

```
StuDataClassesDataContext db=new StuDataClassesDataContext();
db.Class.InsertOnSubmit(c);//插入一条数据
```

现在添加的对象仅仅在内存中，需要执行 SubmitChanges()方法将内存中的更改应用到数据库中：

```
db.SubmitChanges();//提交数据更改到数据库
```

现在查看数据库，可以看到新的班级已经成功添加到数据库中。

使用 LINQ to SQL 修改数据库中的数据时，只需要使用 LINQ 查询找到需要修改的对象，然后修改对象的属性，最后执行 SubmitChanges()方法即可。例如现在需要将学号为 1 的学生姓名修改为“张润”，则对应的程序如代码 11.44 所示。

代码 11.44　使用 LINQ to SQL 修改数据

```
StuDataClassesDataContext db=new StuDataClassesDataContext();
Student student = db.Student.First(s => s.StudentID == 1);
                                              //找到学生 ID 为 1 的学生对象
student.NAME = "张润";                        //修改该学生对象的姓名
db.SubmitChanges();                           //提交修改
```

现在需要将 ClassID 为 3 的班级删除，则使用 LINQ to SQL 删除该行数据的操作如代码 11.45 所示。

代码 11.45　使用 LINQ to SQL 删除数据

```
StuDataClassesDataContext db = new StuDataClassesDataContext();
var cla = from c in db.Class    //找到班级 ID 为 3 的班级对象
     where c.ClassID == 3
     select c;
db.Class.DeleteOnSubmit(cla);   //删除该班级对象
db.SubmitChanges();             //提交删除操作
```

11.8.6　使用 LINQ to SQL 的其他操作

在 LINQ to SQL 中，除了可以使用 LINQ 表达式自动生成 SQL 语句外，还可以直接执行数据库中的存储过程。在 LINQ to SQL 中执行存储过程的操作如下。

（1）创建一个存储过程，通过 StudentID 获得对应的 Student 行，其 SQL 语句如代码 11.46 所示。

代码 11.46　创建存储过程

```
CREATE PROC GetStudentByID --创建一个简单的存储过程
@id INT                    --参数
AS
SELECT *                   --简单的 SQL 查询
FROM Student
WHERE StudentID=@id
```

（2）在 VS 2010 中打开 LINQ to SQL 设计器，将服务器资源管理器中的存储过程拖曳到右侧设计窗口中，如图 11.7 所示。

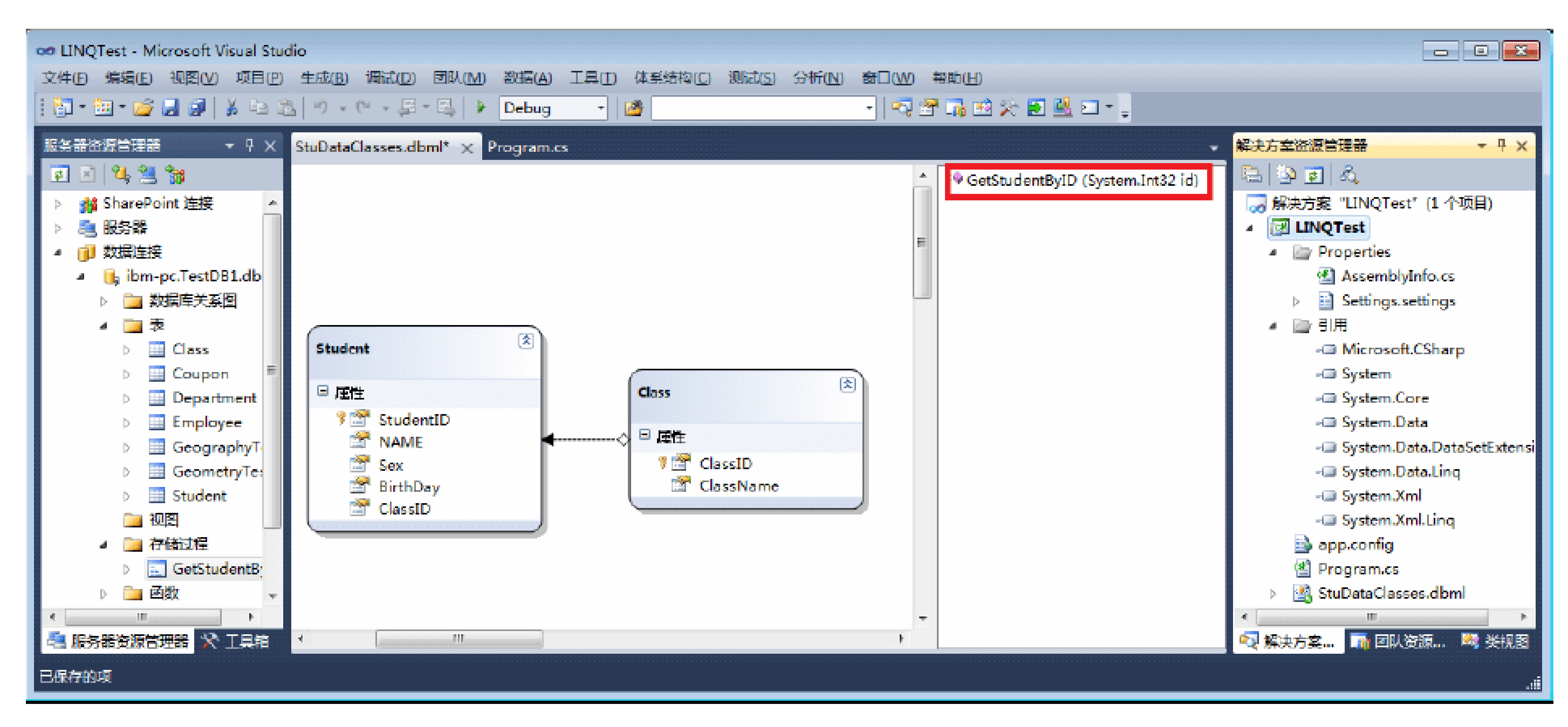


图 11.7　在 LINQ to SQL 设计界面中添加存储过程

（3）保存 LINQ to SQL 文件，系统将会自动创建该存储过程对应的方法。

（4）在程序中调用该存储过程对应的函数并输出返回结果，具体程序如代码 11.47 所示。

代码 11.47　在 LINQ to SQL 中调用存储过程

```
StuDataClassesDataContext db = new StuDataClassesDataContext();
var student = db.GetStudentByID(1);     //调用存储过程对应的方法
foreach (var s in student)
{
    Console.WriteLine(s.NAME);          //输出存储过程返回的结果中的 Name 列
}
```

SQL Server 用户定义函数的添加和使用方法与存储过程相同，这里不再举例。

LINQ to SQL 中支持直接使用 SQL 语句进行数据库查询。查询使用 ExecuteQuery()方法。例如要在数据库执行一条指定的 SQL 语句，返回所有的班级对象，则对应的查询程序如代码 11.48 所示。

代码 11.48　在 LINQ to SQL 中执行 SQL 语句查询

```
StuDataClassesDataContext db = new StuDataClassesDataContext();
var cla = db.ExecuteQuery<Class>("select * from Class");
                                               //定义 SQL 查询语句并执行
foreach (var c in cla)                         //循环输出查询返回的每一个结果
{
    Console.WriteLine(c.ClassID+" "+c.ClassName);
}
```

除了执行数据库查询外，LINQ to SQL 还可以执行数据更改的 SQL 语句。LINQ to SQL 提供的 ExecuteCommand()方法用于指定 SQL 语句执行数据更改，并返回受影响的行数。例如修改 StudentID 为 4 的学生生日，则对应的程序如代码 11.49 所示。

代码 11.49　在 LINQ to SQL 中执行 SQL 语句

```
StuDataClassesDataContext db = new StuDataClassesDataContext();
//以下代码执行数据库修改，并返回受影响的行数
int count=db.ExecuteCommand("update Student set Birthday='1983-12-11'
where StudentID=4");
Console.WriteLine(count);//输出结果：1
```

LINQ to SQL 已经对数据库的操作做了很好的封装，但是为了便于将 LINQ 和其他数据库操作结合起来，实现多个 LINQ 查询的操作，LINQ to SQL 提供了 Connection 属性用于获得当前的连接，Transaction 属性用于指定 LINQ 操作所属的事务。若不使用 SqlTransaction 对象进行事务处理，还可以使用 TransactionScope 用于事务处理。

例如要将 ClassID 为 1 的班级和班级中的所有学生从数据库中删除，则使用 SqlTransaction 结合 LINQ 进行事务处理的程序如代码 11.50 所示。

代码 11.50　在 LINQ to SQL 中使用事务

```
StuDataClassesDataContext db = new StuDataClassesDataContext();
SqlConnection conn = (SqlConnection)db.Connection; //获得连接
conn.Open();                                        //打开连接
SqlTransaction trans = (SqlTransaction)db.Connection.BeginTransaction();
                                                    //创建事务
db.Transaction = trans;                     //将事务赋予 LINQ to SQL 实例 try
{
    db.Student.DeleteAllOnSubmit(from s in db.Student     //删除班级 ID 为 1
                                                            的所有学生
                            where s.ClassID == 1
                            select s);
    db.Class.DeleteAllOnSubmit(from c in db.Class  //删除班级 ID 为 1 的所有班级
                          where c.ClassID == 1
                          select c);
    db.SubmitChanges();                             //提交更改
    trans.Commit();                                 //提交事务
}
catch                                               //发生异常时回滚事务
{
    trans.Rollback();
}
finally                                             //关闭连接
{
    conn.Close();
}
```

11.9　小　　结

本章主要讲解了使用 ADO.NET 对数据库的访问。ADO.NET 是一组向.NET 程序员公开数据访问服务的类。ADO.NET 为创建分布式数据共享应用程序提供了一组丰富的组件，

提供了对关系数据、XML 和应用程序数据的访问。ADO.NET 是从 ADO 发展而来，使用.NET 的程序语言（比如 C#）便可调用。

ADO.NET 中使用数据库连接池管理对数据库的连接。连接池减少新连接需要打开的次数。在创建连接后使用 SqlCommand 对象指定要执行的 SQL 语句或存储过程及需要的参数。ExecuteReader 执行返回行的命令；ExecuteNonQuery 执行命令并返回数据库中受影响的行数；ExecuteScalar 从数据库中检索单个值；ExecuteXmlReader 将 CommandText 发送到 Connection 并生成一个 XmlReader 对象。

通过返回的 SqlDataReader 对象可以从数据库中检索只读、只进的数据流。使用 SqlDataReader 可以编写对应的方法将其转换为实体对象或对象集合。

使用 SqlDataAdapter 可以将 SqlDataReader 对象转换为 DataSet 对象。DataSet 表示整个数据集，其中包含对数据进行包含、排序和约束的表及表间的关系。通过 DataSet 可以获得 DataTable 对象，DataTable 表示一个查询结果集。DataTable 相当于数据库中的表，通过 DataTable 可以获得其中的 DataRow 对象和 DataColumn 对象，分别用来表示行列。可以将 DataTable 转换为 DataView，通过 DataView 对象进行数据的排序和分页。

ADO.NET 中提供了 SqlTransaction 对象和 TransactionScope 对象用于事务处理。SqlTransaction 对象基于一个连接，所以针对的是一个数据库实例，而 TransactionScope 中的事务可以使用多个连接，使用 TransactionScope 可以进行分布式事务处理。

为了简化数据库操作，微软提供了数据访问应用程序块（Data Access Application Block），它包含经过优化的数据访问代码，可以帮助用户调用存储过程及向 SQL Server 数据库发出 SQL 文本命令。它返回 SqlDataReader、DataSet 和 XmlReader 对象。

在.NET Framework 3.5 以上的版本中提供了 LINQ 用于数据处理。在本章中使用了 VS 2010 创建 LINQ to SQL 用于 SQL Server 数据的操作。通过使用 LINQ to SQL，可以在现有数据库架构上直接使用 LINQ 编程模型。通过 LINQ to SQL 可以完成对数据库的增删改操作，也可以调用存储过程、用户定义函数和直接使用 SQL 语句。

第 12 章　使用 SMO 编程管理数据库对象

SMO 是 SQL Server Management Objects 的简称，是微软提供的专门针对 SQL Server 进行编程管理的类库。使用 SMO 在.NET 环境下编程可以轻松实现对 SQL Server 中近乎所有数据库对象的操作。本章将主要介绍 SMO 的基础知识及在 C#中的使用。

12.1　SMO 简介

SMO 是 SQL Server 管理对象的简称，是用来管理 SQL Server 及配置设置信息的对象模型。基于 SMO 的应用程序使用.NET Framework 语言来操纵这个内存中的对象模型，而不用基于同样的目的向 SQL Server 发送 T-SQL 命名。

SMO 在对象模型中封装了 SQL Server 和 T-SQL 各个版本特别的信息，这样使得创建管理 SQL Server 的应用程序非常简单。

SMO 不仅可以管理 SQL Server 数据库引擎，还提供了管理其他 SQL Server 应用程序的功能，例如分析服务、报表服务、数据库作业等。只要 SQL Server Management Studio 能实现的东西，用 SMO 都能实现，因为 SQL Server Management Studio 就是用 SMO 开发的。

与 SMO 相对应的是 ADO.NET，不同的是 ADO.NET 是用于数据访问的，而 SMO 是用于设计的，虽然 SMO 能够在服务器上执行任意的 SQL 语句。另外一个不同的地方是，ADO.NET 可以访问计算机中任意数据源，而 SMO 对象是专门针对 SQL Server 而设计的。

SMO 中最重要也最常用的功能就是对数据库引擎的管理，特别是其中的 Microsoft.SqlServer.Management.SMO 命名空间的类。

SMO 支持 SQL Server 2000、SQL Server 2005、SQL Server 2008 和 SQL Server 2012。SMO 类库是基于.Net Framework 2.0 开发的，所以基于 SMO 的应用程序需要.NET Framework 2.0 来运行。SMO 是 SQL Server 2012 客户端工具的一部分，也是 SQL Server 2012 功能包的一部分。可以在安装目录（例如 C:\Program Files\Microsoft SQL Server\110\SDK\Assembliesr）下找到 SMO 的程序集 Microsoft.SqlServer.Smo.dll。

SMO 把 SQL Server 看做是一个对象集合，每个数据库、表、登录等都被看做是一个对象。基于 SMO 的.NET Framework 2.0 应用程序对这些对象进行编程，来添加或删除数据库、登录、表等。

在 SMO 对象中，运行 SQL Server 的计算机与 SQL Server 本身不同。运行 SQL 服务的计算机由一个 ManagedComputer 类实例来表示，SQL Server 使用 Server 类的实例表示。SMO 有两个独立的对象模型（一个针对运行 SQL Server 应用程序的计算机；另一个针对 SQL Server）和一个运行 SQL Server 2012 中各个服务的 ManagedComputer（SQL Server 本

身、全文搜索等）。另一方面，Server 代表包含数据库和其他对象的数据库引擎。

ManagedComputer 可以用来管理运行 SQL Server 提供服务的机器配置信息。例如，可以用来激活或关闭一个 SQL Server 实例或者改变它的网络配置。作为 SQL Server 客户端工具一部分的 SQL Server 配置管理器，使用 ManagedComputer 激活数据库引擎并管理它的网络配置。

SMO 提供了多个命名空间，分别用于管理数据库引擎、数据库代理、数据库邮件等，具体命名空间和提供的内容如表 12.1 所示。其中最重要的命名空间就是 Microsoft.SqlServer.Management.Smo，本章也主要围绕这个命名空间进行介绍。

表 12.1　SMO中的命名空间

命 名 空 间	内　　容
Microsoft.SqlServer.Management.Smo	SMO 开发核心，包含以编程方式操作 Microsoft SQL Server 的类、实例和枚举
Microsoft.SqlServer.Management.Common	公共基础，包含 RMO 和 SMO 的公共类，比如连接类
Microsoft.SqlServer.Management.Smo.Agent	包含 SQL Server 代理的类
Microsoft.SqlServer.Management.Smo.Wmi	包含 WMI Provider 的类
Microsoft.SqlServer.Management.Smo.RegisteredServers	包含已注册服务器的类
Microsoft.SqlServer.Management.Smo.Mail	包含数据库邮件的类
Microsoft.SqlServer.Management.Smo.Broker	包含 Service Broker 的类

12.2　SMO 对象模型

SMO 为 SQL Server 提供了一个丰富的对象模型。这个对象模型包含 SQL Server 中的许多对象，它的类层次主要由两种对象组成，即 SqlSomObject 对象和 SmoCollectionBase 对象。SqlSmoObject 对象是 Server 对象的父对象。这个对象层次包含开发者感兴趣的大多数对象，如数据库和表。这个层次中的每个对象都有一些属性，每个属性都描述了对象的某些特征或包含指向其他 SqlSomObject 对象或 SmoCollectionBase 对象的引用，它们进一步地扩展了这个层次结构。本节将主要介绍 SMO 对象模型中最常用的几个对象。

12.2.1　SMO 对象和 URN 简介

SMO 对象模型中的对象都是从 SqlSmoObject 继承的类的实例。每个对象都由统一资源名称（Uniform Resource Name，URN）唯一标识，并且包含指向对象层次中其父对象和子对象引用的属性。所有对象都可以使用这些引用来访问，有时通过依次遍历对象树，有时直接通过标识它们的 URN。所有对象都有描述它们的属性，这些属性可以动态或静态地访问。

SMO 对象模型中通过 URN 标识 SQL Server 中的每个 SqlSmoObject。URN 作为持久的、位置无关的资源标识使用。指定对象的 URN 可以通过 SqlSmoObject.Urn 属性获得。SQL Server 中的每个对象都有一个 URN，这是一个很有用的特征。指定一个对象的 URN，

便可以从 Server 对象中直接访问该对象而不用遍历对象树。这意味着如果有一个 SqlSmoObject 对象引用，则可以记住它的 URN，然后使用 URN 直接访问它。

注意：URN 只是 SMO 使用的一个标识方法，它并不是 SQL Server 实例本身的属性。

URN 对 SqlSmoObject 来说很重要，它必须基于一个公式（表示在对象树中从 Server 到该对象的路径）来构造。这意味着可以通过构造 URN 在 SMO 对象树中找到对象。

URN 的语法分为 3 个部分，分别是架构、命名空间标识和特定于命名空间的字符串。URN 的架构总是 urn，并且大小写敏感。以下是一个 URN 的例子。命名空间标识表示该怎样解析第 3 部分的字符串。

```
urn:MS-STUDYZY:OH-Localtion=50/Size=43
```

可以使用构造命名空间标识来跟踪对象。同样地，特定于命名空间的字符串可以是想要的任意格式。在实际操作中，可能使用命名空间标识来判别怎样解析特定于命名空间的字符串。

Server.GetSmoObject()方法可以使用 URN 从 SQL Server 中获取对象。如果 URN 指定的对象存在，则 Server.GetSmoObject()方法返回该对象的引用；否则返回 null。Server.GetSmoObject()方法能够接收格式最规范的 URN，但这不是必要的。它忽略 URN 中除特定于命名空间字符串外的其他所有部分。

特定于命名空间的字符串其实是一个从 SQL Server 中选择对象的 XPath 位置路径。XPath 规范定义了位置路径的语法，可以从 Http://www.w3.org/TR/Xpath 下获得。简单地说，XPath 位置路径由一系列位置步骤组成，相互之间由斜杠（/）分隔。每个位置步骤标识 SMO 维护对象树的一个层次。

XPath 位置路径中的每个位置步骤包含 SMO 维护的 SQL Server 对象种类及该对象的名称。例如，在 MS-STUDYZY 服务器上 SMOTestDB 数据库中，Student 表的特定于命名空间字符串如下。

```
Server[@Name='MS-STUDYZY']/Database[@Name='SMOTestDB']/Table[@Name='Stu
dent' and @Schema='dbo']
```

这个 XPath 位置路径含有 3 个步骤，分别是 Server[@Name='MS-STUDYZY']、Database[@ Name='SMOTestDB']和 Table[@Name='Student' and @Schema='dbo']。

通常，XPath 位置路径忽略额外的空格字符，所以下面的字符串也将标识 Authors 表。XPath 不忽略空格字符的另一个地方是属性值。属性是名称前加@字符，值用双引号括起来，并且名称和值之间使用等号（=）分隔的字符串。下面的特定于命名空间的字符串标识了数据库中的“My Table”表，而不是“MyTable”，即数据库对象的名称中“My”和“Table”之间也必须有一个空格。

```
Server[@Name='CANOPUS5']/Database[@Name='Scratch']/Table[@Name='My Table']
```

XPath 称@Name 为属性，是因为它前面有个@，中括号（[]）中的字符及其他任何东西都必须是一个返回 true 的断言。在前面的例子中，Database[@Name='SMOTestDB']返回所有 Database 对象中 Name 属性值'SMOTestDB'的 Database 对象，当然只有一个。下一个步骤 Table[@Name='Student' and @Schema='dbo']是在前面找到的服务器中查找 Name 属性

为'Student'，并且 Schema 属性为'dbo'的表格，当然也只有一个满足条件。

例如在数据库服务器 MS-STUDYZY 上，通过 URN 获得 AdventureWorks 数据库下的 Person.AddressType 表对应的 SMO 对象 Table，同时输出该对象的名字，对应的 C#代码如代码 12.1 所示。

代码 12.1　通过 URN 获得数据库对象

```
Server server = new Server("MS-STUDYZY");  //声明 Server 对象连接到指定服务器
Urn urn = new Urn("Server[@Name='MS-STUDYZY']/Database[@Name=
'AdventureWorks2012']/Table [@Name='AddressType' and @Schema='Person']");
                                                   //定义一个 URN
Table table =server.GetSmoObject(urn) as Table;    //通过 URN 对象获得对应的
                                                     数据库对象
Console.Write(table.Name);                         //输出数据库对象的名字
```

说明： URN 类在 Microsoft.SqlServer.Management.Sdk.Sfc 命名空间下定义，所以必须要添加对该命名空间的引用才能使用 URN 类。

12.2.2　获得 SMO 对象属性

SMO 对象模型中的所有对象都有一些共有的属性，因为它们都从 SqlSmoObject 继承。其中一个就是 Properties 属性，它是这个对象所有属性的枚举器。

对象的每个属性都由一个 Property 对象描述，Property 对象本身也有 8 个属性，如表 12.2 所示。除了 Type 和 Value 属性，其他都是 System.Boolean 类型，Type 的类型为 System.Type。

表 12.2　Property对象的属性

属 性 名 称	描　　述
Dirty	如果为 true，表示 SQL Server 中的值和这个属性的值可能不同
Expensive	如果为 true，表示加载这个属性需要许多资源
IsNull	表示这个属性的值是否允许为 null
Retrieved	如果为 true，表示已经获取了这个属性的值
Readable	表示是否可以读取这个属性
Type	返回这个属性的 CLR 类型
Value	返回这个属性的值
Writable	如果为 true，表示可以写这个属性

无论是数据库、表还是视图，都可以通过枚举 Properties 获得其所有属性。例如通过 URN 获得一个 Table 对象，输出其属性的代码如代码 12.2 所示。

代码 12.2　获得属性

```
Table table =server.GetSmoObject(urn) as Table;  //通过 URN 获得一个表对象
foreach(Property p in  table.Properties)         //循环获得表对象下的所有属性
{
    Console.WriteLine(p.Name+":"+p.Value);       //输出属性名和属性值
}
```

SMO 对象层次从 Server 类开始。这个对象模型构建在继承有 SqlSmoObject 或 SmoCollectionBase 的属性对象之上。一个 SqlSmoObject 属性保存一个子对象的引用，SmoCollectionBase 属性保存子对象的引用集合。

注意：完整的 SMO 类图非常大，类中的属性也很多。本节只侧重介绍其中很小的一部分，以说明在 SMO 中怎样浏览对象。

在 SMO 对象模型中，除了 Server，每个 SqlSmoObject 对象都有一个 Parent 属性，这个属性保存一个到其父对象的引用，并且和它的父对象类型相同。例如，Server.ServiceMasterKey 是一个到 ServerMasterKey 对象的引用，在 SMO 对象模型中，它是 Server 对象的子对象。ServiceMasterKey.Parent 是一个到 Server 类型对象的引用，后者是它的父对象。

SmoCollectionBase 对象保存的子对象 Parent 属性所指的对象是保存该集合的对象，而不是集合对象本身。例如，Server.Databases 是一个保存 Database 对象的集合对象。Database 对象的 Database.Parent 所指的是 Server 对象，而不是 Database 集合对象。

大多数 SMO 集合对象都有一个关联的索引器，可以用它根据名称或位置来访问集合中的对象。例如，如果集合中包含 AdventureWorks2012 数据库，Databases["AdventureWorks2012"]将返回一个到该数据库对象的引用，否则返回 null。包含在架构中的对象集合的索引器额外有一个可选的参数（用于标识架构名）。如果没有指定它，将使用连接到服务器的登录名的默认架构。

SqlSmoObject.State 表示对象的状态。SqlSmoObject 的状态值是 SqlSmoState 枚举值中的一个。如表 12.3 所示为 SqlSmoState 的所有值。

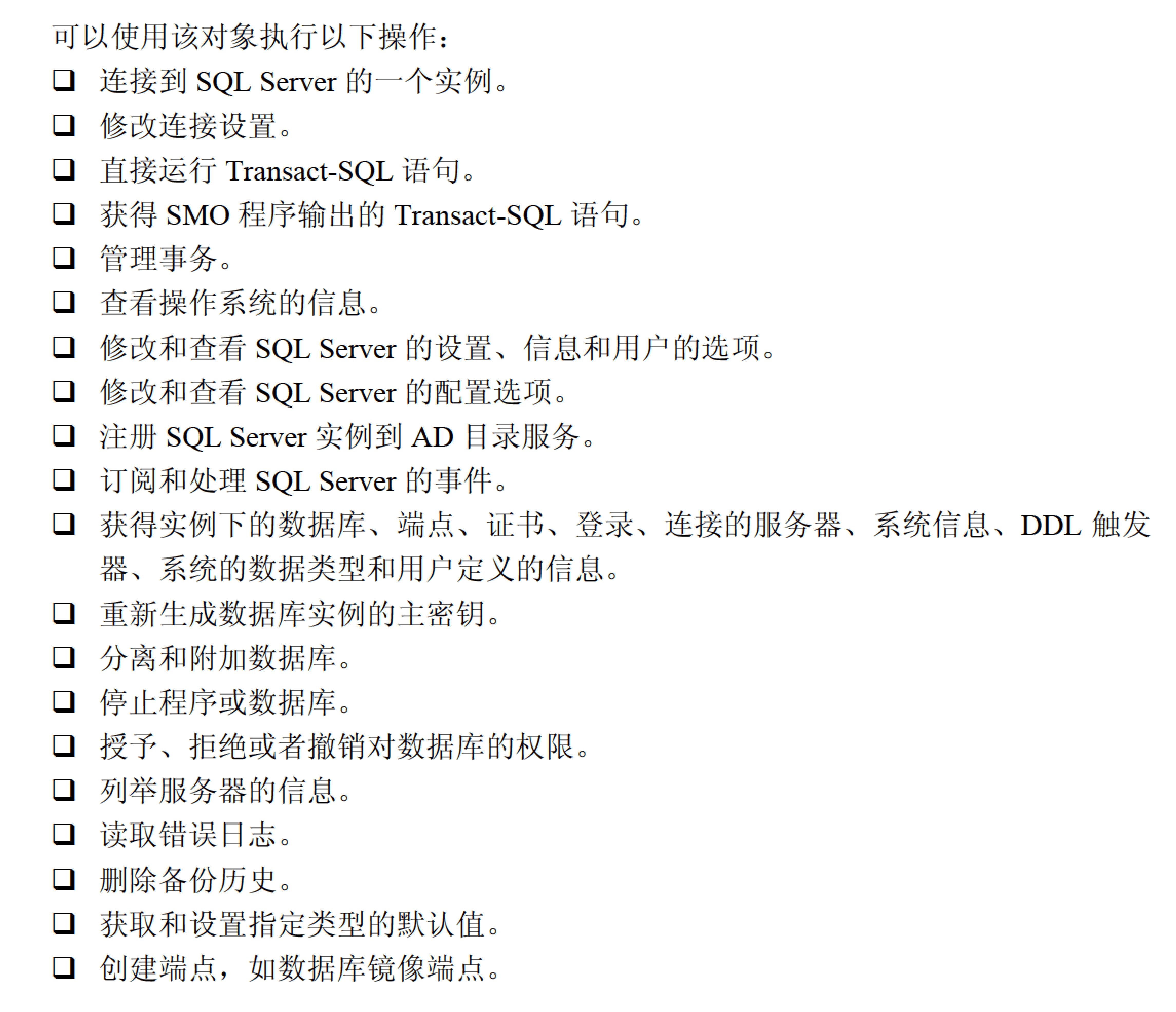
表 12.3　SqlSmoState的值

SqlSmoState 值	描　　述
Creating	正在创建
Dropped	正在删除
Existing	存在
Pending	对象正在创建过程中，所以它的 Name 或 Parent 等字段还没有被填充
ToBeDropped	等待删除

12.2.3　Server 对象简介

sqlserver.exe 是 SQL Server 的可执行程序，作为 Windows 服务来运行。可以同时运行 sqlserver.exe 的多个实例，在这种情况下每个服务都有不同的名称。在任何情况下，每个实例都被 SMO 看做是一个 Server 对象。每个 SQL Server 实例可以包含多个数据库，每个数据库可以包含多个表格等。通常，服务的名称和运行它的机器相同。Server 对象提供了以下 3 个构造函数。

- Server()：使用 Windows 身份认证为本机默认的数据库实例构造 Server 对象。
- Server(ServerConnection)：通过向 ServerConnection 对象中传递服务器地址、用户名和密码来构造指定的对象。
- Server(String)：使用 Windows 身份认证为指定的数据库实例构造对象。



可以使用该对象执行以下操作：

- ❑ 连接到 SQL Server 的一个实例。
- ❑ 修改连接设置。
- ❑ 直接运行 Transact-SQL 语句。
- ❑ 获得 SMO 程序输出的 Transact-SQL 语句。
- ❑ 管理事务。
- ❑ 查看操作系统的信息。
- ❑ 修改和查看 SQL Server 的设置、信息和用户的选项。
- ❑ 修改和查看 SQL Server 的配置选项。
- ❑ 注册 SQL Server 实例到 AD 目录服务。
- ❑ 订阅和处理 SQL Server 的事件。
- ❑ 获得实例下的数据库、端点、证书、登录、连接的服务器、系统信息、DDL 触发器、系统的数据类型和用户定义的信息。
- ❑ 重新生成数据库实例的主密钥。
- ❑ 分离和附加数据库。
- ❑ 停止程序或数据库。
- ❑ 授予、拒绝或者撤销对数据库的权限。
- ❑ 列举服务器的信息。
- ❑ 读取错误日志。
- ❑ 删除备份历史。
- ❑ 获取和设置指定类型的默认值。
- ❑ 创建端点，如数据库镜像端点。

注意：要获得或者设置数据库实例的信息，需要连接数据库实例的用户具有相应的权限。

12.2.4　Database 对象简介

Database 对象对应的就是数据库实例中的一个具体数据库或数据库快照，这其中也包括系统数据库。通过 Server 对象的 Databases 属性，可以获得数据库实例中的 Database 对象集合。可以通过索引或者数据库名从 Database 集合中找到需要的数据库对象。例如要找到 MS-STUDYZY 服务器上 AdventureWorks2012 数据库对应的 Database 对象，则对应的 C#代码如代码 12.3 所示。

代码 12.3　通过数据库名获得 Database 对象

```
Server server = new Server("MS-STUDYZY");              //声明 Server 对象
Database db = server.Databases["AdventureWorks2012"];//通过 Server 对象获得
                                                      其中的 Database 对象
```

通过使用 Database 对象，可以做到以下几点：

- ❑ 创建一个新的数据库或删除现有的数据库。
- ❑ 注册数据库到 AD 目录服务。

- ❑ 获得数据库对象的集合，例如表、用户和视图等。
- ❑ 安装数据库镜像。
- ❑ 创建一个数据库的主密钥。
- ❑ 安装全文搜索目录。
- ❑ 检查数据、配额、目录和表。
- ❑ 重大问题的检查。
- ❑ 授予、拒绝和撤销数据库中用户的权限。
- ❑ 运行 T-SQL 语句。
- ❑ 枚举数据库信息，如锁或对象的权限。
- ❑ 删除备份的历史。
- ❑ 监视事务的次数。
- ❑ 数据库设置为脱机或联机。
- ❑ 改变数据库的所有者。
- ❑ 更新统计信息。
- ❑ 收缩数据库。
- ❑ 截断日志。
- ❑ 为数据库生成脚本。

12.2.5　Table 对象简介

Table 对象对应的是数据库中的表。通过 Database 对象的 Tables 属性可以获得数据库中所有表的集合，其中也包括系统表。通过索引或表名和架构可以获得指定的 Table 对象。例如要获得 AdventureWorks2012 数据库中 Person.AddressType 表对应的 Table 对象，则对应的 C#程序如代码 12.4 所示。

代码 12.4　通过架构和表名获得 Table 对象

```
Server server = new Server("MS-STUDYZY");
Database db = server.Databases["AdventureWorks2012"];
Table table = db.Tables["AddressType", "Person"];//获得 Person.AddressType
                                                  表对应的 Table 对象
```

对于默认架构的表，则可以直接通过表名来获得 Table 对象，例如要获得 AdventureWorks2012 数据库中的 dbo.DatabaseLog 表所对应的 Table 对象，由于当前用户的默认架构是 dbo，所以可以不提供架构而直接提供表名，如代码 12.5 所示。

代码 12.5　直接通过表名获得 Table 对象

```
Server server = new Server("MS-STUDYZY");
Database db = server.Databases["AdventureWorks2012"];
Table logTable = db.Tables["DatabaseLog"];//获得 DatabaseLog 对应的 Table 对象
```

与 Server 对象和 Database 对象相似，通过 Table 对象也可以：

- ❑ 创建、修改和删除表。

- ❑ 获得表下的数据库对象，例如列、约束、索引等。
- ❑ 获得或设置表的权限、扩展等属性。
- ❑ 为表编写脚本。

12.2.6　其他对象简介

View 对象对应数据库中的视图，通过 Database 对象的 Views 属性可以获得数据库中所有视图包括系统视图的集合。与 Table 对象的获取相同，View 对象可以通过索引或通过视图名和架构获得。例如，要获得 AdventureWorks2012 数据库的 HumanResources.vEmployee 视图对应的 View 对象，则对应的 C#代码如代码 12.6 所示。

代码 12.6　通过架构和视图名获得 View 对象

```
Server server = new Server("MS-STUDYZY");
Database db = server.Databases["AdventureWorks2012"];
View view = db.Views["vEmployee","HumanResources"];//获得数据库下的视图对象
```

StoredProcedure 对象对应数据库中的存储过程，通过 Database 对象的 StoredProcedures 属性，可以获得数据库中所有存储过程的集合，包括系统存储过程。StoredProcedure 对象也是通过在 StoredProcedures 属性中提供索引或通过存储过程名和架构获得。例如要获得 AdventureWorks2012 数据库的存储过程 dbo. uspGetManagerEmployees 对应的 StoredProcedure 对象，则对应的 C#代码如代码 12.7 所示。

代码 12.7　通过存储过程名获得 StoredProcedure 对象

```
Server server = new Server("MS-STUDYZY");
Database db = server.Databases["AdventureWorks2012"];
StoredProcedure sp = db.StoredProcedures["uspGetManagerEmployees"];//获得
数据库中的存储过程对象
```

说明：由于 dbo 架构是当前用户的默认架构，所以可以直接通过存储过程名获得 StoredProcedure 对象，而无须提供架构名。

对于表来说，最重要的对象就是其中的列，Column 对象对应的就是数据库中表或视图的列。通过 Table 对象的 Columns 属性可以获得表列的集合。Column 对象可以通过索引或列名从 Columns 属性中获得。例如要获得 AdventureWorks2012 数据库中 Person. AddressType 表 Name 列对应的 Column 对象，则对应的 C#代码如代码 12.8 所示。

代码 12.8　通过列名获得 Column 对象

```
Server server = new Server("MS-STUDYZY");
Database db = server.Databases["AdventureWorks2012"];
Table table = db.Tables["AddressType", "Person"]; //获得数据库下的 Table 对象
Column col = table.Columns["Name"];              //获得表对象下的 Column 对象
```

在获得了 Column 对象后，就可以通过属性获得该对象对应列的数据类型、是否允许为空、默认值、是否为标识列、是否为主键等。

通过 SMO 对象模型，还可以获得函数、触发器、登录名、用户等数据库对象的信息。即使是一个十分简单的数据库，其对应的整个 SMO 对象模型仍然是十分庞大的。

12.3　创建 SMO 应用程序

SMO 应用程序需要.Net Framework 2.0 的支持，使用 VS 2005 以上版本都可以开发 SMO 应用程序。本节将基于 C# 4.5 的语法，以控制台应用程序示例的方式，说明 SMO 对象模型的使用和应用程序的创建。

12.3.1　在 VS 中创建 SMO 项目

由于 VS 中没有专门 SMO 项目的模板，所以需要手动做好项目的创建、程序集的引用等操作。以创建 SMO 的控制台应用程序为例，在 VS 创建 SMO 项目的主要操作如下。

（1）打开 VS 2012，选择“创建项目”选项，弹出“新建项目”对话框，如图 12.1 所示。

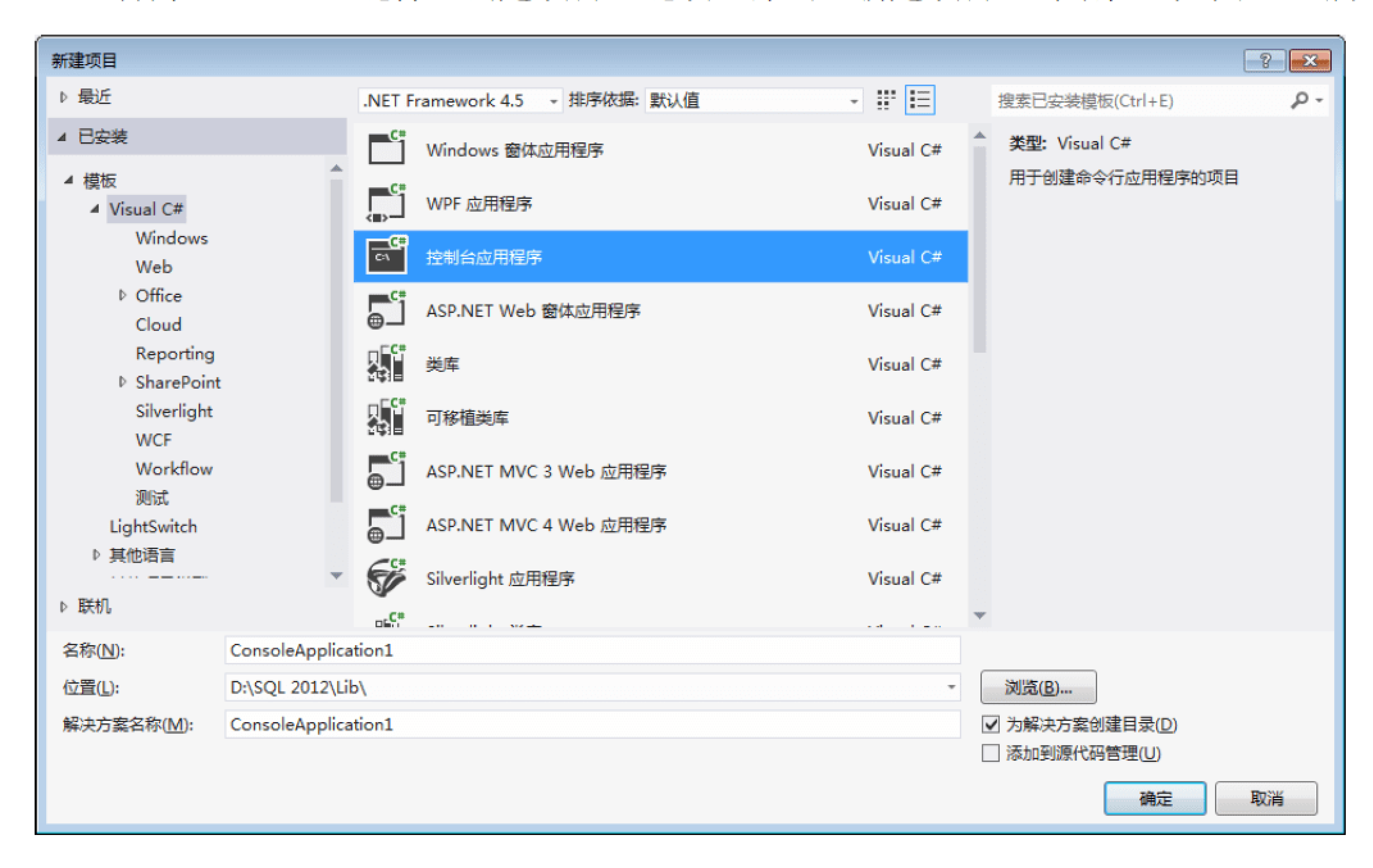


图 12.1　“新建项目”对话框

（2）在左侧项目类型中选择 Visual C# 选项，右侧模板中选择“控制台应用程序”选项。由于.NET Framework 4.5/3.5 与 2.0 兼容，所以随便选择哪个版本都可以，这里选择 2.0 版。使用默认项目名称 ConsoleApplication1，单击“确定”按钮，系统将创建控制台应用程序项目，其创建的解决方案如图 12.2 所示。

（3）由于 SMO 编程中所使用的命名空间 Microsoft.SqlServer.Smo 不在系统默认引用的程序集中，所以需要添加对 SMO 程序集的引用。右击 ConsoleApplication1 项目，在弹出的快捷菜单中选择“添加引用”选项，弹出“引用管理器”对话框，如图 12.3 所示。在右上角输入要查找的命名空间的名称，即可查找到所需的命名空间。

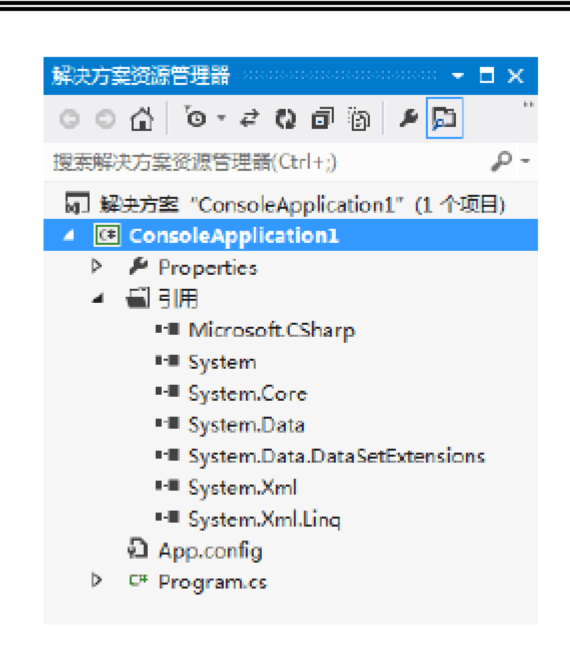


图 12.2　控制台应用程序的解决方案

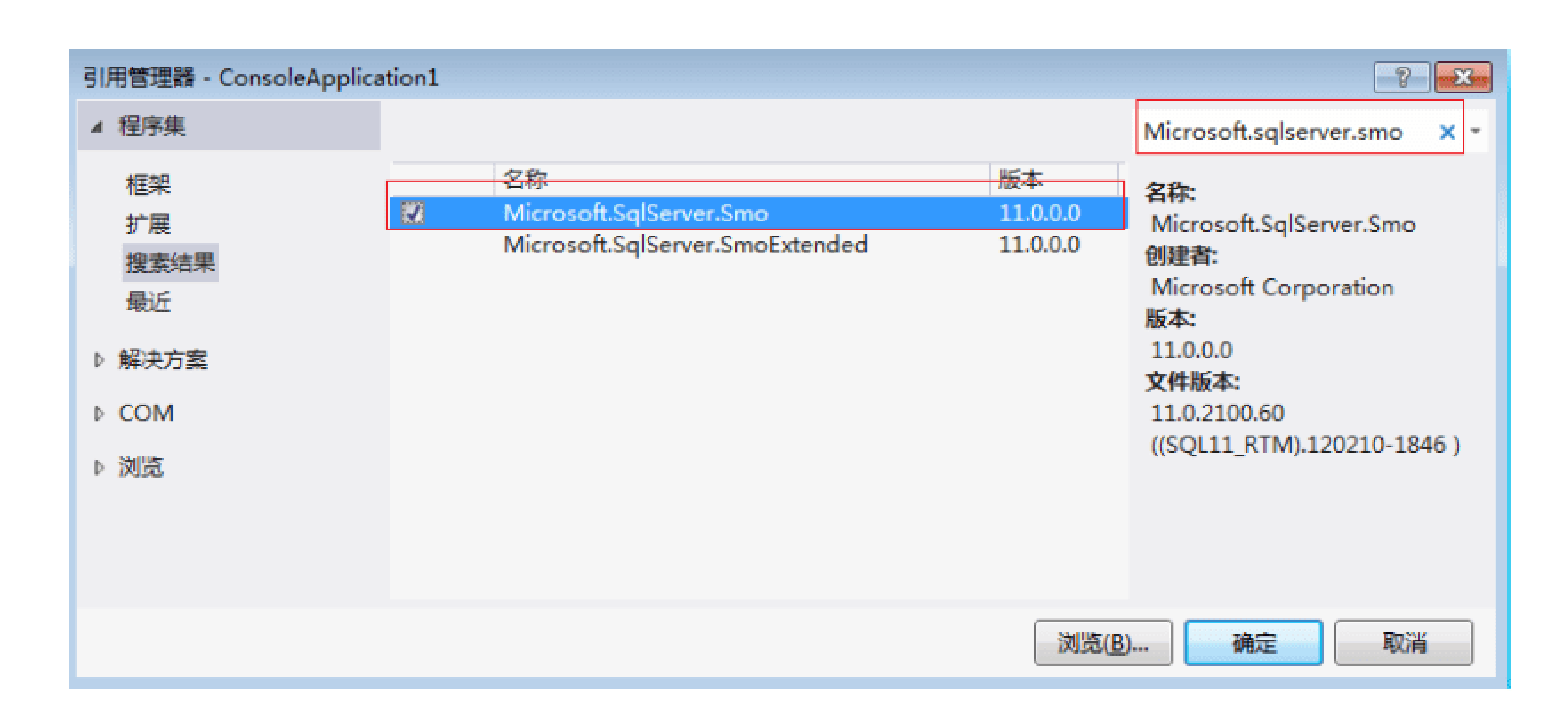


图 12.3 “引用管理器”对话框

（4）在 SMO 编程中总共需要用到 3 个程序集，分别是 Microsoft.SqlServer.ConnectionInfo、Microsoft.SqlServer.Smo 和 Microsoft.SqlServer.Management.Sdk.Sfc。在“引用管理器”对话框中选中这 3 个程序集，然后单击“确定”按钮，系统将会把这 3 个程序集添加到引用中，通过解决方案资源管理器可以看到当前项目引用的程序集，如图 12.4 所示。

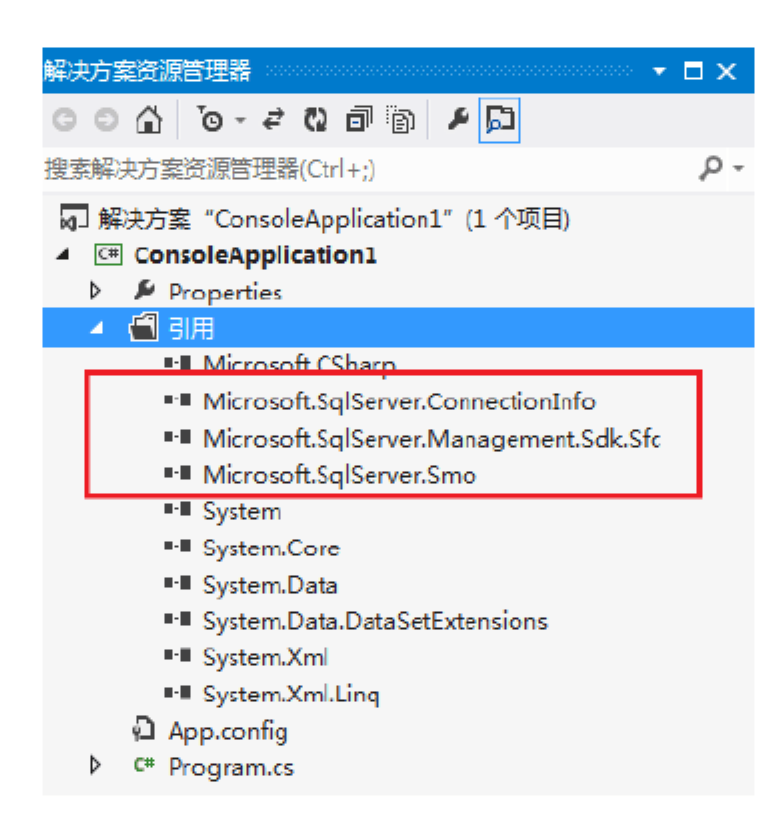


图 12.4　查看已经添加的引用

注意：如果同时安装了 SQL Server 2008 和 SQL Server 2012，则将会看到两个相同名字的程序集，一个是 10.0 版，另外一个是 11.0 版。由于本书是针对 SQL Server 2012 的 SMO 编程，所以选择 11.0 版即可。

（5）在程序中添加对应 SMO 命名空间的引用，使用 using 语句即可，如代码 12.9 所示。

代码 12.9　添加命名空间

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.SqlServer.Management.Smo;      //需要添加的 SMO 命名空间
using Microsoft.SqlServer.Management.Common;   //需要添加的命名空间
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

至此，SMO 编程的准备工作已经完成，接下来就可以根据实际需要编写代码了。

12.3.2　使用 SMO 管理数据库

在 SMO 中对数据库的管理都是通过 Database 对象来完成，下面通过几个示例来说明 SMO 编程在数据库管理中的应用。

1. 创建数据库

现在有数据库服务器 192.168.100.102，现需要使用 SMO 编程在其中创建数据库 SMOTestDB，则主要思路和代码如下：

（1）使用具有创建数据库权限的用户（例如 sa）连接到数据库服务器，创建 Server 对象实例。

```
//以下代码通过 IP 用户名和密码获得数据库连接对象
ServerConnection conn = new ServerConnection("192.168.100.102", "sa",
"p@ssw0rd");
//通过数据库连接对象获得 Server 对象
Server server = new Server(conn);
```

（2）创建一个 Database 对象，并将其命名为 SMOTestDB。

```
Database db = new Database(server, "SMOTestDB");//获得具体的数据库对象
```

（3）将当前对象模型中创建的 Database 应用到数据库中。

```
db.Create();//创建数据库
```

总共只编写了 4 行代码，在 SQL Server 中创建数据库的操作就完成了。完整的创建数据库的示例如代码 12.10 所示。

代码 12.10　创建数据库的示例

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.SqlServer.Management.Smo;        //SMO 所在的命名空间
using Microsoft.SqlServer.Management.Common;    //SMO 所在的命名空间
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        //通过 IP 用户名和密码获得连接对象
         ServerConnection conn = new ServerConnection("192.168.100.102",
         "sa", "p@ssw0rd");
            Server server = new Server(conn);//获得 Server 对象
        //通过数据库名获得 Database 对象,但是该数据库并不存在
            Database db = new Database(server, "SMOTestDB");
            db.Create();                                //创建数据库
        }
    }
}
```

2. 获得数据库信息

对于现有数据库，可以通过 SMO 获得数据库的属性，也可以通过 SMO 获得数据库下的表、架构、视图、用户、存储过程等数据库对象。

例如对于 AdventureWorks2012 数据库，要获取该数据库的一些属性和数据库对象，则主要思路和代码为：

（1）连接到服务器获得 Server 对象。

```
ServerConnection conn = new ServerConnection("192.168.100.102", "sa",
"p@ssw0rd");
Server server = new Server(conn);
```

（2）获得 AdventureWorks 对应的 Database 对象。

```
Database db = server.Databases["AdventureWorks2012"];
```

（3）根据 Database 对象获得 AdventureWorks 数据库的大小。

```
Console.WriteLine("当前数据库大小："+db.Size+" MB"); //获得 Database 的 Size
                                                      属性
```

（4）根据 Database 对象获得 AdventureWorks2012 数据库的创建日期、最后备份日期。

```
Console.WriteLine("创建日期："+db.CreateDate);   //获得 Database 对象的属性
Console.WriteLine("最后备份日期：" + db.LastBackupDate);
```

（5）根据 Database 对象获得该数据库中所有的架构并输出。

```
foreach (Schema schema in db.Schemas)  //循环输出数据库中的所有架构名
{
    Console.WriteLine("拥有架构：" + schema.Name);
}
```

（6）使用同样的方法可以获得数据库下的所有表、视图、存储过程等数据库对象，这里不再举例。

3．修改数据库

在 SMO 编程中修改数据库的属性后，并不会将对象的修改应用到数据库中。Database 对象下提供了 Alter()方法，用于将 Database 对象的更改更新到对应的数据库中。

例如要修改 AdventureWorks2012 数据库，将该数据库设置为只读的，则对应的代码如代码 12.11 所示。

代码 12.11　修改数据库

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.SqlServer.Management.Smo;       //SMO 所在的命名空间
using Microsoft.SqlServer.Management.Common;    //SMO 所在的命名空间
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
         ServerConnection conn = new ServerConnection("192.168.100.102",
         "sa", "p@ssw0rd");
            Server server = new Server(conn);
            Database db = server.Databases["AdventureWorks2012"];
                                                //通过名字获得 Database 对象
            db.ReadOnly = true;                 //将数据库设置为只读
            db.Alter();                         //应用修改到数据库服务器中
        }
    }
}
```

再将 ReadOnly 属性设置为 false 并重新执行一次便可去掉数据库只读的限制。

4．删除数据库

若要删除数据库，SMO 为 Database 对象提供了 Drop()方法。例如要将前面创建的 SMOTestDB 数据库删除，则对应的代码如代码 12.12 所示。

代码 12.12　删除数据库

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.SqlServer.Management.Smo;       //SMO 所在的命名空间
using Microsoft.SqlServer.Management.Common;    //SMO 所在的命名空间
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
         ServerConnection conn = new ServerConnection("192.168.100.102",
```

```
        "sa", "p@ssw0rd");
          Server server = new Server(conn);
          Database db = server.Databases["SMOTestDB"];//通过数据库名获得
                                                       Database 对象
          db.Drop();                                  //删除该数据库
        }
    }
}
```

在 SMO 编程中，数据库的创建、修改和删除对应的 Create()、Alter()和 Drop()方法，与 SQL 语句创建、修改和删除数据库相同。

12.3.3　使用 SMO 管理表

在 SMO 对象模型中，Table 对象与 SQL Server 中的表对应，通过对 Table 对象的操作实现了对数据库表的管理。下面以几个示例来说明如何使用 SMO 管理表。

1．创建表

首先通过 SSMS 或者使用 SMO 在服务器上创建一个数据库 SMOTestDB，新创建的数据库除了系统表外没有任何用户表。现要在其中创建一个学生表 Student，该表包含了学生的基本信息：学号 StudentID、学生姓名 Name、性别 Sex、出生日期 Birthday 等字段。使用 SMO 编程创建该学生表的步骤和代码如下。

（1）连接到数据库服务器获得 Server 对象。

```
ServerConnection conn = new ServerConnection("192.168.100.102", "sa",
"p@ssw0rd");
Server server = new Server(conn);
```

（2）获得 SMOTestDB 数据库对应的 Database 对象。

```
Database db = server.Databases["SMOTestDB"];
```

（3）创建 Table 对象，并将 Table 命名为 Student。

```
Table stuTable = new Table(db, "Student"); //创建 Table 对象
```

（4）创建 StudentID 的 Column 对象，设置该对象不允许为空。

```
                                   //创建列对象
Column studentID = new Column(stuTable, "StudentID", DataType.Char(10));
studentID.Nullable = false;
stuTable.Columns.Add(studentID);   //将列添加到表中
```

（5）用同样的方法创建 Name、Sex 和 Birthday 对应的 Column 对象。

```
Column name = new Column(stuTable, "Name", DataType.NVarChar(10));
name.Nullable = false;
stuTable.Columns.Add(name);        //创建列 Name 并添加到表中
Column sex = new Column(stuTable, "Sex", DataType.Bit);
sex.Nullable = false;
stuTable.Columns.Add(sex);         //创建列 Sex 并添加到表中
Column birthday = new Column(stuTable, "Birthday", DataType.Date);
birthday.Nullable = false;
stuTable.Columns.Add(birthday);    //创建列 Birthday 并添加到表中
```

（6）创建该表的主键 PK_StudentID。

```
Index pk = new Index(stuTable, "PK_StudentID");    //创建索引对象
pk.IndexKeyType = IndexKeyType.DriPrimaryKey;      //索引类型是主键
```

注意：要使用枚举 IndexKeyType.DriPrimaryKey，则必须要添加对程序集 Microsoft.SqlServer.SqlEnum 的引用。

（7）设置主键 PK_StudentID 所在的列 StudentID。

```
pk.IndexedColumns.Add(new IndexedColumn(pk, "StudentID"));
```

（8）将创建的主键索引添加到表 Student 对象中。

```
stuTable.Indexes.Add(pk);
```

（9）使用 Table 对象下的 Create 方法，将对象模型中的更改提交到数据库中。

```
stuTable.Create();
```

编译运行以上 SMO 程序，使用 SMO 对表的创建已经完成，通过 SSMS 连接到数据库，可以看到该表已经被创建成功。

2. 修改表

与修改数据库类似，修改表可以使用 Table 对象下的 Alter()方法或者 AlterWithNoCheck()方法。这两个方法的区别在于：Alter()方法需要检查更改的属性值而 AlterWithNoCheck()方法则不检查。同样以前面创建的 Student 表为例，现在若需要在该表中添加地址字段 Address，则对应的 C#代码如代码 12.13 所示。

代码 12.13　修改表

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.SqlServer.Management.Smo;      //SMO 所在的命名空间
using Microsoft.SqlServer.Management.Common;   //SMO 所在的命名空间
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
         ServerConnection conn = new ServerConnection("192.168.100.102",
         "sa", "p@ssw0rd");
            Server server = new Server(conn);
            Database db = server.Databases["SMOTestDB"];//通过数据库名获得
                                                    Database 对象
            Table stuTable = db.Tables["Student"];   //通过表名获得 Table 对象
            //创建新的列 Address
            Column address = new Column(stuTable, "Address", DataType.
            NVarChar (200));
            address.Nullable = false;
            stuTable.Columns.Add(address);           //将列 Address 添加到表中
```

```
            stuTable.Alter();                          //应用对表添加列的修改
        }
    }
}
```

3．删除表和列

删除表或者删除表中的列都使用 Drop()方法。删除列与 T-SQL 中删除列不同的是，在 SMO 中删除列并不认为是对 Table 对象的修改，而是对 Column 的删除，所以不需要执行 Table 的 Alter()方法。例如要删除表 Student 中的列 Address，则对应的代码如代码 12.14 所示。

代码 12.14　删除列

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.SqlServer.Management.Smo;      //SMO 所需的命名空间
using Microsoft.SqlServer.Management.Common;   //SMO 所需的命名空间
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
         ServerConnection conn = new ServerConnection("192.168.100.102",
         "sa", "p@ssw0rd");
            Server server = new Server(conn);
            Database db = server.Databases["SMOTestDB"];//通过数据库名获得
                                                        Database 对象
            db.Tables["Student"].Columns["Address"].Drop();
            //通过表名和列名获得列对象，然后删除该列
        }
    }
}
```

删除整个表则使用 Table 对象的 Drop()方法：

```
db.Tables["Student"].Drop();
```

12.3.4　使用 SMO 管理存储过程

相对数据库和表来说，存储过程的管理有所不同，但是 SMO 中仍然提供 Create()、Alter()和 Drop()方法进行添加、修改和删除。

1．创建存储过程

存储过程在 SMO 中对应的对象为 StoredProcedure，该对象提供了 Parameters 属性用于设置存储过程传入的参数；提供了 TextBody 属性用于设置存储过程的内容。

以创建一个存储过程 Hello 为例，该存储过程接收参数@name，运行该存储过程就为该参数打招呼。具体创建存储过程的步骤如下。

（1）连接到数据库服务器获得 Server 对象。

```
ServerConnection conn = new ServerConnection("192.168.100.102", "sa",
"p@ssw0rd");
Server server = new Server(conn);
```

（2）获得 SMOTestDB 数据库对应的 Database 对象。

```
Database db = server.Databases["SMOTestDB"];
```

（3）创建 StoredProcedure 对象，并为该对象命名为 Hello。

```
StoredProcedure sp=new StoredProcedure(db,"Hello");
```

（4）将 StoredProcedure 对象的 TextMode 属性设置为 false。

```
sp.TextMode = false;
```

注意：这个属性至关重要，如果没有设置，接下来创建存储过程时会报错。

（5）为存储过程添加参数@name。

```
sp.Parameters.Add(new StoredProcedureParameter(sp,"@name",DataType.Nvar-
Char(10)));
```

（6）设置存储过程的内容。

```
sp.TextBody = "SELECT 'Hello '+@name as Hello";
```

（7）使用 Create()方法将存储过程创建到数据库中。

```
sp.Create();
```

运行该 SMO 程序，便可将存储过程 Hello 创建到数据库中。另外，如果需要设置存储过程之前的 ANSI_NULLS 属性和 QUOTED_IDENTIFIER 属性等，可以在第（4）步通过修改 sp 的属性来实现。

```
sp.AnsiNullsStatus = false;
sp.QuotedIdentifierStatus = false;
```

2．修改和删除存储过程

修改存储过程的主要思路是通过存储过程名获得对应的 StoredProcedure 对象，然后对该对象的属性进行更改，修改完成后执行 Alter()方法即可。

例如要修改前面新建的存储过程 Hello，传入两个参数，同时向两个人打招呼，则对应的修改代码如代码 12.15 所示。

代码 12.15　修改存储过程

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.SqlServer.Management.Smo;      //SMO 所需的命名空间
using Microsoft.SqlServer.Management.Common;   //SMO 所需的命名空间
namespace ConsoleApplication1
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            ServerConnection conn =
            new ServerConnection("192.168.100.102", "sa", "p@ssw0rd");
            Server server = new Server(conn);
            Database db = server.Databases["SMOTestDB"]; //通过数据库名获得
                                                          Database 对象
            StoredProcedure sp = db.StoredProcedures["Hello"];
            //通过存储过程名获得 StoredProcedure 对象
            sp.TextMode = false;
            sp.Parameters.Add(new StoredProcedureParameter(sp,
            "@name2", DataType.NVarChar(10)));  //为存储过程添加参数
            sp.TextBody = "SELECT 'Hello '+@name+','+@name2";
                                                //设置存储过程内容
            sp.Alter();                         //应用对存储过程的修改到数据库
        }
    }
}
```

注意：即使在创建时设置了 TextMode 为 false，在修改该存储过程时仍然需要重新设置 TextMode 为 false。

删除存储过程只需要使用 Drop()方法即可：

```
sp.Drop();
```

12.3.5　使用 SMO 生成脚本

在 SMO 中对数据库的操作，实际上将会被转换为 T-SQL 脚本用来创建、修改和删除 SQL Server 中的对象。可以使用 SMO 来创建这些脚本。SMO 还能管理对象依赖性，以使脚本中对象的创建和删除能以一定的次序进行。

SMO 对象有一个 Script()方法，这个方法用来生成可以创建或删除相应对象的 T-SQL 脚本。默认情况下 Script()方法生成创建对象的脚本，脚本作为 StringCollection 返回。注意 StringCollection 类在 System.Collections.Specialized 命名空间中。

例如要生成一个创建 AdventureWorks 数据库中 Person.AddressType 表的脚本。使用 Script()方法返回一个 StringCollection 命名脚本。脚本中的字符串通过 foreach 语句枚举并打印到控制台，如代码 12.16 所示。

代码 12.16　生成创建表的脚本

```
ServerConnection  conn  =  new  ServerConnection("192.168.100.102",  "sa",
"p@ssw0rd");
Server server = new Server(conn);
Database db = server.Databases["AdventureWorks2012"];  //获得 Database 对象
Table table=db.Tables["AddressType","Person"]; //获得 Table 对象
foreach (string script in table.Script())      //循环输出创建该表的完整脚本
{
    Console.WriteLine(script);
}
```

系统将生成创建表的脚本：

```
SET ANSI NULLS ON
SET QUOTED IDENTIFIER ON
CREATE TABLE [Person].[AddressType](
       [AddressTypeID] [int] IDENTITY(1,1) NOT NULL,
       [Name] [dbo].[Name] NOT NULL,
       [rowguid] [uniqueidentifier] ROWGUIDCOL  NOT NULL,
       [ModifiedDate] [datetime] NOT NULL
) ON [PRIMARY]
```

可以给 Script()方法传入一个 ScriptingOptions 对象，以控制脚本的生成方式。有许多可用的脚本选项，例如创建了一个 ScriptOptions 对象，其 ScriptDrops 和 IncludeIfNotExists 属性被设置为 true，然后把它作为参数传给 Script()方法。具体代码和生成的脚本如代码 12.17 所示。

代码 12.17　生成删除脚本

```
ServerConnection conn = new ServerConnection("192.168.100.102", "sa",
"p@ssw0rd");
Server server = new Server(conn);
Database db = server.Databases["AdventureWorks2012"]; //获得 Database 对象
Table table=db.Tables["AddressType","Person"];     //获得 Table 对象
ScriptingOptions options = new ScriptingOptions(); //创建 ScriptingOptions
                                                     对象
options.IncludeIfNotExists = true; //是否包含如果不存在则执行某操作的脚本
options.ScriptDrops = true;                        //是否删除脚本
foreach (string script in table.Script(options))   //循环输出删除表的脚本
{
    Console.WriteLine(script);
}
```

系统生成的脚本：

```
IF  EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'
[Person].[AddressType]') AND type in (N'U'))
DROP TABLE [Person].[AddressType]
```

生成脚本的大部分工作都涉及多个对象，多个对象的脚本通常比单个对象大得多。Scripter 对象可以为一个对象列表生成脚本，并把脚本直接保存到文件而不是 String Collection。

Scripter 对象需要 Server 对象来生成脚本。可以在 Scripter 构造函数中把 Server 对象作为参数传入，也可以在生成脚本之前把 Server 对象设为它的 Server 属性。Scripter 可以为 SMO 对象数组、URN 或 UrnCollection 生成脚本。要把脚本输出到文件，必须设置 Scripter.Options.FileName 和 ToFileOnly 属性。注意，这些技巧也可以用来为单个对象生成脚本。

Scripter.Script()方法调用后将创建一个名为 FileName 的文件，脚本也被写到这个文件中，如果 ScripterOptions.AppendToFile 为 true，则脚本将被添加到文件末尾。例如要将 AddressType 的创建脚本生成到 D 盘中，使用 Scripter 生成脚本的代码如代码 12.18 所示。

代码 12.18　使用 Scripter 生成脚本到文件

```
ServerConnection conn = new ServerConnection("192.168.100.102", "sa",
"p@ssw0rd");
```

```
Server server = new Server(conn);                  //创建 Server 对象
Scripter sper = new Scripter(server);              //通过Server对象创建Scripter对象
UrnCollection urns = new UrnCollection();
Urn urn = new Urn("Server[@Name='MS-STUDYZY']/Database[@Name='Adventure
Works2012']/Table [@Name='AddressType' and @Schema='Person']");
                                                   //定义 Urn
urns.Add(urn);
sper.Options.ToFileOnly = true;                    //将脚本生成到文件
sper.Options.FileName="D:\\Script.sql";            //指定文件路径
sper.Script(urns);                                 //生成脚本
```

有时候，某些对象依赖于其他对象，脚本要考虑到这点。在调用 Script()方法之前把 Scripter.Options.WithDependencies 属性设为 true，这将使对象以适当的顺序生成脚本，具体取决于生成的是创建脚本，还是删除脚本。另外，它还包含列表中对象的所有依赖对象。

12.4　小　　结

SMO 是管理 SQL Server 的应用程序使用的理想类库。虽然从表面上看，它是用来构建 DBA 类型应用程序的，但事实上它对所有需要 SQL Server 管理功能的应用程序都有用。

SMO 和 VS 紧密集成，并提供了一个等同于 VS 其他工具的编程模型。它对 SQL Server 进行了封装，所以无须知道 SQL Server 的数据库架构及 T-SQL 语言。

SMO 高效地管理着它的对象模型以尽量减小对 SQL Server 和应用程序的影响。对于特别复杂需要消耗大量资源才能获得的属性，则是在需要时才加载，并且能把操作组合在一个往返行程中。

SMO 能够产生管理 SQL Server 的 T-SQL 脚本。这些脚本考虑了对象的依赖关系并可以跟踪用于管理 SQL Server 的操作。

第 13 章　高级 T-SQL 语法

随着数据库技术的发展，使得对数据库的操作不仅仅是简单的增删改查。为了满足不同用户的需求，SQL Server 在每个新版本中都增强了 T-SQL 语句的功能，增加了 T-SQL 语法。本章将主要讲解 SQL Server 2005、SQL Server 2008 及 2012 这 3 个版本中新增的 T-SQL 语法。

13.1　SQL Server 2005 新增语法

SQL Server 2005 相对 SQL Server 2000 来说改动较大，新增的语法也较多，很多新增语法在实际项目中得到了广泛的应用，本节就主要讲解 SQL Server 2005 中的新增语法。

13.1.1　排名函数

在 SQL Server 2005 中引入了称为排名函数的一类新函数。构成该类的函数有 RANK()、DENSE_RANK()、NTILE()和 ROW_NUMBER()函数。排名函数为分区中的每一行返回一个排名值。

1．RANK()函数

RANK()函数返回结果集分区内每行的排名。行的排名是相关行之前的排名数加 1。其语法格式如下：

```
RANK ( )    OVER ( [ < partition_by_clause > ] < order_by_clause > )
```

其中，<partition_by_clause>将 FROM 子句生成的结果集划分为要应用 RANK()函数的分区。<order_by_clause>确定将 RANK 值应用于分区中的行时所基于的顺序。例如有一个学生成绩表，创建并初始化该表的脚本如代码 13.1 所示。

代码 13.1　创建并初始化 Student 表

```
CREATE TABLE Student
(
    StudentID int,   --学生 id
    ClassID int,     --班级编号
    Mark int         --成绩
);
GO
--接下来插入示例数据
INSERT INTO Student VALUES(1,1,90);
```

```
INSERT INTO Student VALUES(2,1,84);
INSERT INTO Student VALUES(3,1,80);
INSERT INTO Student VALUES(4,1,80);
INSERT INTO Student VALUES(5,1,90);
INSERT INTO Student VALUES(6,1,76);
INSERT INTO Student VALUES(7,1,89);
INSERT INTO Student VALUES(11,2,90);
INSERT INTO Student VALUES(12,2,82);
INSERT INTO Student VALUES(13,2,80);
INSERT INTO Student VALUES(14,2,80);
INSERT INTO Student VALUES(15,2,90);
INSERT INTO Student VALUES(16,2,75);
INSERT INTO Student VALUES(17,2,89);
```

通过 RANK()函数可以得到学生成绩在每个班的排名，具体排名脚本如代码 13.2 所示。

代码 13.2　使用 RANK()函数排序

```
SELECT *
   ,RANK() OVER(                     --使用 RANK()函数进行排名
   PARTITION BY ClassID              --使用 ClassID 进行分组
   ORDER BY Mark DESC) AS [Rank]     --使用 Mark 进行排序
FROM Student
```

系统返回结果如表 13.1 所示。

表 13.1　RANK()函数返回的结果

StudentID	ClassID	Mark	Rank	StudentID	ClassID	Mark	Rank
1	1	90	1	15	2	90	1
5	1	90	1	11	2	90	1
7	1	89	3	17	2	89	3
2	1	84	4	12	2	82	4
3	1	80	5	13	2	80	5
4	1	80	5	14	2	80	5
6	1	76	7	16	2	75	7

从结果可以看出，使用 RANK()函数对每个班的分数进行排序后，如果有相同的排名，则接下来的排名将直接跳过顺序值。例如表 13.1 中的结果，有 2 个第 1 名，接下来 RANK()函数返回的是第 3 名，而不是第 2 名。

2．DENSE_RANK()函数

DENSE_RANK()函数返回结果集分区中行的排名，在排名中没有任何间断。行的排名等于所讨论行之前的所有排名数加 1。其语法格式为：

```
DENSE_RANK ( )    OVER ( [ < partition_by_clause > ] < order_by_clause > )
```

参数<partition_by_clause>将 FROM 子句所生成的结果集划分为数个将应用 DENSE_RANK()函数的分区。<order_by_clause>确定将 DENSE_RANK()函数值应用于分区中各行的顺序。同样以前面创建的学生表为例，如果使用 DENSE_RANK()函数对每个班级进行排名，则对应的脚本如代码 13.3 所示。

代码 13.3　使用 DENSE_RANK()函数进行排名

```
SELECT *
    ,DENSE_RANK()                          --使用 DENSE_RANK()函数进行排名
    OVER(PARTITION BY ClassID              --使用 ClassID 进行分组
    ORDER BY Mark DESC) AS [Rank]          --依据 Mark 列排序
FROM Student
```

系统返回的结果如表 13.2 所示。

表 13.2　DENSE_RANK()函数返回的排名结果

StudentID	ClassID	Mark	Rank	StudentID	ClassID	Mark	Rank
1	1	90	1	15	2	90	1
5	1	90	1	11	2	90	1
7	1	89	2	17	2	89	2
2	1	84	3	12	2	82	3
3	1	80	4	13	2	80	4
4	1	80	4	14	2	80	4
6	1	76	5	16	2	75	5

将表 13.2 与 RANK()函数返回的结果表 13.1 进行对比，可以很容易地看出，DENSE_RANK()函数返回的排名是连续的，出现了 2 个第 1 名后，接下来的名次是第 2 名。

3. NTILE()函数

NTILE()函数将有序分区中的行分发到指定数目的组中。各个组有编号，编号从 1 开始。对于每一个行，NTILE()函数将返回此行所属组的编号。该函数的语法格式为：

```
NTILE (integer_expression) OVER ( [ <partition_by_clause> ] < order_by_
clause > )
```

其参数 integer_expression 是一个正整数常量，用于指定每个分区必须被划分成的存储桶的数量。integer_expression 的类型可以为 bigint。<partition_by_clause>将 FROM 子句生成的结果集划分成 RANK()函数适用的分区。<order_by_clause>确定 NTILE()函数值分配到分区中各行的顺序。

如果分区的行数不能被 expression 整除，则将导致一个成员有两种大小不同的组。按照 OVER 子句指定的顺序，较大的组排在较小的组前面。例如，如果总行数是 53，存储桶数是 5，则前 3 个存储桶每个包含 11 行，其余两个存储桶每个包含 10 行。另一方面，如果总行数可被存储桶数整除，则行数将在存储桶之间平均分布。例如，如果总行数为 50，有 5 个存储桶，则每个存储桶将包含 10 行。同样以前面的学生表为例，假设现在要根据考试成绩分快慢班，分成 2 个班，则使用 NTILE()函数进行分班的脚本如代码 13.4 所示。

代码 13.4　使用 NTILE()函数进行分组

```
SELECT *
    ,NTILE(2) OVER(ORDER BY Mark DESC) AS NewClass
```

```
                                                    --使用 NTILE 函数对 Mark 排名
FROM Student
```

系统返回结果如表 13.3 所示。

表 13.3　使用NTILE()函数分组结果

StudentID	ClassID	Mark	NewClass	StudentID	ClassID	Mark	NewClass
1	1	90	1	12	2	82	2
11	2	90	1	13	2	80	2
5	1	90	1	14	2	80	2
15	2	90	1	3	1	80	2
7	1	89	1	4	1	80	2
17	2	89	1	6	1	76	2
2	1	84	1	16	2	75	2

4．ROW_NUMBER()函数

ROW_NUMBER()函数是排名函数中实际使用最广泛的。该函数一般在项目中用于数据库分页。

ROW_NUMBER()函数返回结果集分区内行的序列号，每个分区的第一行从 1 开始。ROW_NUMBER()函数的语法格式为：

```
ROW_NUMBER ( )    OVER ( [ <partition_by_clause> ] <order_by_clause> )
```

其中参数<partition_by_clause>将 FROM 子句生成的结果集划入应用了 ROW_NUMBER()函数的分区。<order_by_clause>确定将 ROW_NUMBER()值分配给分区中的行的顺序。例如对于前面的学生表，如果不存在并列排名，则使用 ROW_NUMBER()函数获得所有学生的年级排名的脚本如代码 13.5 所示。

代码 13.5　使用 ROW_NUMBER()函数获得排名

```
SELECT *
    ,ROW_NUMBER ( )  OVER(ORDER BY Mark DESC) AS OrderID
                                                    --使用函数对 Mark 进行排名
FROM Student
```

系统返回的结果如表 13.4 所示。

表 13.4　使用ROW_NUMBER()函数排名的结果

StudentID	ClassID	Mark	OrderID	StudentID	ClassID	Mark	OrderID
1	1	90	1	12	2	82	8
11	2	90	2	13	2	80	9
5	1	90	3	14	2	80	10
15	2	90	4	3	1	80	11
7	1	89	5	4	1	80	12
17	2	89	6	6	1	76	13
2	1	84	7	16	2	75	14

说明： 使用 ROW_NUMBER()函数进行数据库分页一般与公用表表达式一起使用，在介绍公用表表达式时再讲解如何使用 ROW_NUMBER()函数进行数据库分页。

13.1.2　异常处理

在 SQL Server 2005 中，对 Transact-SQL 实现类似于 C#和 C++语言中异常处理的错误处理。Transact-SQL 语句组可以包含在 TRY 块中。如果 TRY 块内部发生错误，则会将控制传递给 CATCH 块中包含的另一个语句组。异常处理的语法如代码 13.6 所示。

代码 13.6　异常处理语法

```
BEGIN TRY
    { sql statement | statement block }
END TRY
BEGIN CATCH
    { sql statement | statement block }
END CATCH
```

TRY…CATCH 构造捕捉所有严重级别大于 10，但不终止数据库连接的错误。TRY 块后必须紧跟相关联的 CATCH 块。在 END TRY 和 BEGIN CATCH 语句之间放置任何其他语句，都将生成语法错误。

TRY…CATCH 构造不能跨越多个批处理，也不能跨越多个 Transact-SQL 语句块。例如，TRY…CATCH 构造不能跨越 Transact-SQL 语句的两个 BEGIN…END 块，且不能跨越 IF…ELSE 构造。

如果 TRY 块所包含的代码中没有错误，则当 TRY 块中最后一个语句完成时，会将控制传递给紧跟在相关联的 END CATCH 语句之后的语句。如果 TRY 块所包含的代码中有错误，则会将控制传递给相关联的 CATCH 块的第一个语句。如果 END CATCH 语句是存储过程或触发器的最后一个语句，则会将控制传递回调用存储过程或触发器的语句。

当 CATCH 块中的代码完成时，会将控制传递给紧跟在 END CATCH 语句之后的语句。由 CATCH 块捕获的错误不会返回到调用的应用程序。如果任何错误消息都必须返回到应用程序，则 CATCH 块中的代码必须使用 SELECT 结果集，或 RAISERROR 和 PRINT 语句之类的机制执行此操作。

TRY…CATCH 构造可以是嵌套式的。TRY 块或 CATCH 块均可包含嵌套的 TRY…CATCH 构造。例如，CATCH 块可以包含内嵌的 TRY…CATCH 构造，以处理 CATCH 代码所遇到的错误。

处理 CATCH 块中遇到错误的方法与处理任何其他位置生成的错误一样。如果 CATCH 块包含嵌套的 TRY…CATCH 构造，则嵌套的 TRY 块中的任何错误都会将控制传递给嵌套的 CATCH 块。如果没有嵌套的 TRY…CATCH 构造，则会将错误传递回调用方。

TRY…CATCH 构造可以从存储过程或触发器（由 TRY 块中的代码执行）捕捉未处理的错误。存储过程或触发器也可以包含其自身的 TRY…CATCH 构造，以处理由其代码生成的错误。例如，当 TRY 块执行存储过程且存储过程中发生错误时，可以使用以下方式处理：

- 错误会将控制返回与包含 EXECUTE 语句的 TRY 块相关联的 CATCH 块。
- 如果存储过程包含 TRY…CATCH 构造，则错误会将控制传输给存储过程中的

CATCH 块。当 CATCH 块代码完成时，控制会传递回调用存储过程的 EXECUTE 语句之后的语句。

不能使用 GOTO 语句输入 TRY 或 CATCH 块，也不能使用 GOTO 语句跳转至同一个 TRY 或 CATCH 块内的某个标签，或离开 TRY 或 CATCH 块。

不能在用户定义函数内使用 TRY…CATCH 构造。

在 CATCH 块的作用域内，可以使用以下系统函数来获取导致 CATCH 块执行的错误消息。

- ERROR_NUMBER()：返回错误号。
- ERROR_SEVERITY()：返回严重性。
- ERROR_STATE()：返回错误状态号。
- ERROR_PROCEDURE()：返回出现错误的存储过程或触发器的名称。
- ERROR_LINE()：返回导致错误的例程中的行号。
- ERROR_MESSAGE()：返回错误消息的完整文本。该文本可包括任何可替换参数所提供的值，如长度、对象名或时间。

如果是在 CATCH 块的作用域之外调用这些函数，则这些函数返回空值。可以从 CATCH 块作用域内的任何位置使用这些函数检索错误消息。例如创建一个表 t1，然后试着向其中插入重复主键的数据，从而导致异常发生，可以通过 TRY…CATCH 构造来处理发生的异常，具体脚本如代码 13.7 所示。

代码 13.7　TRY…CATCH 处理异常

```
CREATE TABLE t1
(
    c1 int PRIMARY KEY,
    c2 varchar(50)
)
GO
INSERT INTO t1 VALUES(1,'good');    --插入测试数据
GO
--以下异常处理代码单独执行：
BEGIN TRY
    INSERT INTO t1 VALUES(1,'same')
END TRY
BEGIN CATCH                         --捕捉到异常后进行处理
    SELECT ERROR_LINE(),ERROR_SEVERITY(),ERROR_MESSAGE()   --输出异常内容
END CATCH
```

选择运行异常处理代码，系统将返回如下错误信息。

```
2 24 违反了 PRIMARY KEY 约束 'PK__t1__3213663B59FA5E80'。不能在对象 'dbo.t1'
中插入重复键。
```

TRY…CATCH 构造在下列情况下不捕获错误。

- 严重性为 10 或更低的警告或信息性消息。
- 严重性为 20 或更高且终止会话的 SQLServerDatabaseEngine 任务处理的错误。如果所发生错误的严重性为 20 或更高，而数据库连接未中断，则 TRY…CATCH 将处理该错误。
- 需要关注的消息，如客户端中断请求或客户端连接中断。

- 当系统管理员使用 KILL 语句终止会话时。

如果以下类型错误的发生级别与 TRY…CATCH 构造的执行等级相同，则 CATCH 块不会处理这些错误。

- 编译错误，例如防止执行批处理的语法错误。
- 语句级重新编译过程中出现的错误，例如由于名称解析延迟而造成在编译后出现的对象名称解析错误。

这些错误会被返回到运行批处理、存储过程或触发器的级别。

如果某个错误在 TRY 块内的编译或语句级别重新编译的过程中，并在较低的执行级别（例如，执行 sp_executesql 或用户定义存储过程时）处发生，则该错误会在低于 TRY…CATCH 构造的级别上发生，并由相关联的 CATCH 块处理。

13.1.3　APPLY 操作符

使用 APPLY 运算符可以为实现查询操作的外部表表达式返回每个行调用表值函数。表值函数作为右输入，外部表表达式作为左输入。通过对右输入求值来获得左输入每一行的计算结果，生成的行被组合起来作为最终输出。APPLY 运算符生成列的列表是左输入中的列集，后跟右输入返回列的列表。

APPLY 有两种形式，即 CROSS APPLY 和 OUTER APPLY。CROSS APPLY 仅返回外部表中通过表值函数生成结果集的行。OUTER APPLY 既返回生成结果集的行，也返回不生成结果集的行，其中表值函数生成的列中的值为 NULL。

说明：APPLY 运算符可以认为是一个表和另一个表值函数的 JION 操作。

SQL Server 提供了动态管理函数 sys.dm_exec_sql_text(sql_handle)，该函数接收一个 SQL 语句的句柄，返回一个表，该表包含了 SQL 语句的文本、执行的数据库 ID、是否加密等信息。动态管理视图 sys.dm_exec_cached_plans 提供了当前缓存中执行计划的 SQL 句柄，通过将该视图和表值函数 sys.dm_exec_sql_text(sql_handle)执行 APPLY 操作，便可获得缓存中执行计划的 SQL 文本，具体脚本如代码 13.8 所示。

代码 13.8　使用 APPLY

```
SELECT p.usecounts, p.cacheobjtype, p.objtype, s.text
FROM sys.dm exec cached plans p
CROSS APPLY sys.dm exec sql text(plan handle) s    --一个表与表值函数进行
                                                     APPLY 操作
```

13.1.4　PIVOT 和 UNPIVOT 运算符

新的 PIVOT 和 UNPIVOT 运算符对结果集进行旋转，这样，列变成了行，行变成了列。这称为旋转数据或创建交叉报表。在早期版本的 SQL Server 中可以进行这些操作，但需要复杂逻辑，现在通过使用 PIVOT 和 UNPIVOT 这一逻辑得到了简化。例如一个产品销售表，该表记录了每个产品每个季度的销量，创建并初始化该表如代码 13.9 所示。

代码 13.9　创建并初始化产品销售表

```
CREATE TABLE ProductSale   --创建测试表和测试数据
(
    ID int,
    Name varchar(20),
    Quarter int,
    Sale int
)
insert into ProductSale values(1,'a',1,1000)
insert into ProductSale values(1,'a',2,2000)
insert into ProductSale values(1,'a',3,4000)
insert into ProductSale values(1,'a',4,5000)
insert into ProductSale values(2,'b',1,3000)
insert into ProductSale values(2,'b',2,3500)
insert into ProductSale values(2,'b',3,4200)
insert into ProductSale values(2,'b',4,5500)
```

现在需要统计每一个商品每个季度的销量，由于季度是在行中，需要将季度作为列输出，所以需要使用 PIVOT 运算符。使用 PIVOT 运算符获得产品每个季度销量的脚本如代码 13.10 所示。

代码 13.10　使用 PIVOT 行转列

```
SELECT ID,Name,
[1] as "一季度",
[2] as "二季度",
[3] as "三季度",
[4] as "四季度"
FROM
ProductSale
PIVOT   --进行行转列操作
(
sum(Sale)
for Quarter in
([1],[2],[3],[4])
)
as pvt
```

系统返回的结果为：

```
  ID       Name     一季度    二季度    三季度     四季度
-------  ------  ------  -------  -------  -------
  1        a      1000     2000     4000      5000
  2        b      3000     3500     4200      5500
```

现在以该结果的形式创建 ProducteSale2，将数据写入该表，具体脚本如代码 13.11 所示。

代码 13.11　创建并初始化表

```
CREATE TABLE ProductSale2      --创建测试表和测试数据
(
    ID int,
    Name varchar(20),
    Q1 int, Q2 int, Q3 int, Q4 int
)
insert into ProductSale2 values(1,'a',1000,2000,4000,5000)
insert into ProductSale2 values(2,'b',3000,3500,4200,5500)
```

对于该表，若希望将季度作为列，而每个季度的值作为行输出查询结果，则需要使用 UNPIVOT 运算符，具体脚本如代码 13.12 所示。

代码 13.12　使用 UNPIVOT

```
SELECT ID,Name,Quarter,Sale
FROM ProductSale2
UNPIVOT --使用 UNPIVOT 列转行
(
Sale
for Quarter in
([Q1],[Q2],[Q3],[Q4])
)
as unpvt
```

系统将返回的结果与表 ProductSale 相同。

13.1.5　OUTPUT 语法

OUTPUT 子句返回受 INSERT、UPDATE 或 DELETE 语句影响的每行的信息，或者返回基于上述每行的表达式。这些结果可以返回到处理应用程序，以供在确认消息、存档及其他类似的应用程序要求中使用。此外，也可以将结果插入表或表变量。OUTPUT 子句的语法格式如代码 13.13 所示。

代码 13.13　OUTPUT 语法

```
<OUTPUT CLAUSE> ::=
{
    [ OUTPUT <dml select list> INTO { @table variable | output table }
[ ( column list ) ] ]
    [ OUTPUT <dml select list> ]
}
<dml_select_list> ::=
{ <column_name> | scalar_expression } [ [AS] column_alias_identifier ]
    [ ,...n ]

<column_name> ::=
{ DELETED | INSERTED | from_table_name } . { * | column_name }
```

其中各参数的主要意思是：

- @table_variable 指定一个 table 变量，返回的行将插入此变量，而不是返回给调用方。@table_variable 必须在 INSERT、UPDATE、DELETE 或 MERGE 语句前声明。
- output_table 指定一个表，返回的行将被插入该表中而不是返回到调用方。output_table 可以为临时表。如果未指定 column_list，则表必须与 OUTPUT 结果集具有相同的列数。标识列和计算列例外，必须跳过这两种列。如果指定了 column_list，则任何省略的列都必须允许为 Null 值，或者都分配有默认值。output_table 无法应用于以下情况：具有启用的对其定义的触发器；参与 FOREIGNKEY 约束的任意一方；具有 CHECK 约束或启用的规则。
- column_list 是 INTO 子句目标表上列名的可选列表。它类似于 INSERT 语句中允许

使用列的列表。

- scalar_expression 可取计算结果为单个值的任何符号和运算符的组合。scalar_expression 中不允许使用聚合函数。对修改的表中列的任何引用，都必须使用 INSERTED 或 DELETED 前缀限定。
- column_alias_identifier 用于引用列名的代替名称。
- DELETED 指定由更新或删除操作删除的值的列前缀。以 DELETED 为前缀的列，反映了 UPDATE、DELETE 或 MERGE 语句完成之前的值。不能在 INSERT 语句中同时使用 DELETED 与 OUTPUT 子句。
- INSERTED 列的前缀，指定由插入或更新操作添加的值。以 INSERTED 为前缀的列，反映了在 UPDATE、INSERT 或 MERGE 语句完成之后，但在触发器执行之前的值。INSERTED 语句不能与 DELETE 语句的 OUTPUT 子句同时使用。
- from_table_name 是一个列前缀，指定 DELETE、UPDATE 或 MERGE 语句（用于指定要更新或删除的行）的 FROM 子句中包含的表。如果还在 FROM 子句中指定了要修改的表，则对该表中的列的任何引用都必须使用 INSERTED 或 DELETED 前缀限定。指定受删除、插入或更新操作影响的所有列，都将按照它们在表中的顺序返回。
- column_name 显式列引用。对要修改的表的任何引用，都必须根据需要使用 INSERTED 或 DELETED 前缀正确限定，例如，INSERTED.column_name。
- $ACTION 仅可用于 MERGE 语句。在 MERGE 语句的 OUTPUT 子句中指定一个 nvarchar(10)类型的列，该子句为每行返回以下三个值之一：'INSERT'、'UPDATE'或'DELETE'，返回哪个值取决于对该行执行的操作。

OUTPUT<dml_select_list>子句和 OUTPUT <dml_select_list> INTO { @table_variable | output_table }子句，可以在单个 INSERT、UPDATE 或 DELETE 语句中定义。除非另行指定，否则，对 OUTPUT 子句的引用将同时引用 OUTPUT 和 OUTPUTINTO 子句。

OUTPUT 子句对于在 INSERT 或 UPDATE 操作之后，检索标识列或计算列的值可能非常有用。当<dml_select_list>中包含计算列时，输出表或表变量中的相应列并不是计算列。新列中的值是在执行该语句时计算出的值。以下语句中不支持 OUTPUT 子句。

- 引用本地分区视图、分布式分区视图或远程表的 DML 语句。
- 包含 EXECUTE 语句的 INSERT 语句。
- 当数据库兼容级别设为 100 时，不允许在 OUTPUT 子句中使用全文谓词。
- 不能将 OUTPUT INTO 子句插入视图或行集函数。
- 如果用户定义的函数包含一个以表为目标的 OUTPUT INTO 子句，则不能创建该函数。不能将 OUTPUT INTO 子句插入视图或行集函数。

如果将参数或变量作为 UPDATE 语句的一部分进行了修改，则 OUTPUT 子句将始终返回语句执行之前的参数或变量的值，而不是已修改的值。在使用 WHERECURRENTOF 语法通过游标定位的 UPDATE 或 DELETE 语句中，可以使用 OUTPUT。

OUTPUT 子句支持下列大型对象数据类型：nvarchar(max)、varchar(max)、varbinary(max)、text、ntext、image 和 xml。当在 UPDATE 语句中使用.WRITE 子句修改 nvarchar(max)、varchar(max)或 varbinary(max)列时，如果引用了值的全部前像和后像，则将其返回。在 OUTPUT 子句中，TEXTPTR()函数不能作为 text、ntext 或 image 列的表达式

的一部分出现。

例如要删除一些数据，同时希望将删除的数据返回给程序，则对应的脚本如代码 13.14 所示。

代码 13.14　使用 OUTPUT 显示删除的数据

```
CREATE TABLE Student         --创建测试表
(
   StudentID int,            --学生 id
   ClassID int,              --班级编号
   Mark int                  --成绩
);
GO
--创建测试数据
INSERT INTO Student VALUES(1,1,90);
INSERT INTO Student VALUES(2,1,84);
INSERT INTO Student VALUES(3,1,80);
INSERT INTO Student VALUES(4,1,80);
INSERT INTO Student VALUES(5,1,90);
INSERT INTO Student VALUES(6,1,76);
INSERT INTO Student VALUES(7,1,89);
INSERT INTO Student VALUES(11,2,90);
INSERT INTO Student VALUES(12,2,82);
INSERT INTO Student VALUES(13,2,80);
INSERT INTO Student VALUES(14,2,80);
INSERT INTO Student VALUES(15,2,90);
INSERT INTO Student VALUES(16,2,75);
INSERT INTO Student VALUES(17,2,89);
--以上是初始化数据，接下来要删除一些数据
DELETE FROM Student
OUTPUT deleted.*     --将删除的数据输出
WHERE StudentID = 1;
```

现在已经从 Student 表中删除了学号为 1 的学生的数据，同时将会把删除的数据返回到客户端，输出结果为：

```
StudentID   ClassID     Mark
1              1          90
```

现在再向该表中插入一条数据，同时在插入完成后将插入的数据输出，使用 OUTPUT 实现的脚本如代码 13.15 所示。

代码 13.15　OUTPUT 输出 INSERT 结果

```
INSERT INTO Student
OUTPUT INSERTED.*    --输出插入的行
VALUES(1,1,91)
```

系统将输出结果：

```
StudentID   ClassID     Mark
1              1          91
```

现在发现插入的数据不对，需要将 91 分改为 90 分，那么需要使用 UPDATE 语句。若需要将 UPDATE 的内容输出，则可以使用 OUTPUT 子句，具体脚本如代码 13.16 所示。

代码 13.16　OUTPUT 输出 UPDATE 结果

```
UPDATE Student
SET Mark=90
OUTPUT
DELETED.StudentID AS OldStudentID,      --输出更新操作时原来的数据
DELETED.ClassID AS OldClassID,
DELETED.Mark AS OldMark,
INSERTED.StudentID AS NewStudentID,     --输出更新后的数据
INSERTED.ClassID AS NewClassID,
INSERTED.Mark AS NewMark
WHERE StudentID=1
```

系统将输出结果：

```
OldStudentID   OldClassID  OldMark  NewStudentID   NewClassID  NewMark
1                   1         91         1              1          90
```

与 UPDATE 触发器一样，OUTPUT 子句中认为更新操作是由删除操作和插入操作完成，所以可以在 UPDATE 操作中使用 DELETED 和 INSERTED 表。

除了输出删除或插入的数据外，OUTPUT 另一个重要的作用是将数据进行备份。将删除的数据移动到另外一个表中，这样在操作或者数据发生错误时可以通过备份的已删除数据查找相关信息。

同样以这里使用的 Student 表为例，现需要建立一个 StudentDeleted 表，该表保存完整的被删除的记录，则对应的删除操作如代码 13.17 所示。

代码 13.17　删除数据到备份表

```
CREATE TABLE StudentDeleted
(
   StudentID int,   --学生 id
   ClassID int,     --班级编号
   Mark int         --成绩
);
GO
DELETE FROM Student
OUTPUT deleted.* INTO StudentDeleted
--将数据从 Student 表删除，删除的数据插入到 StudentDeleted 表中
WHERE StudentID = 1;
```

13.1.6　公用表表达式 CTE

公用表表达式（CTE）是在 SELECT、INSERT、UPDATE 或 DELETE 语句执行过程中暂时存储的结果集。利用 CTE 可使用递归查询，并可通过取代临时表或视图来简化逻辑。

可以将公用表表达式(CTE)视为临时结果集，在 SELECT、INSERT、UPDATE、DELETE 或 CREATEVIEW 语句的执行范围内进行定义。CTE 与派生表类似，具体表现在不存储为对象，并且只在查询期间有效。与派生表的不同之处在于，CTE 可自引用，还可在同一查询中引用多次。CET 可用于：

- 创建递归查询。
- 在不需要常规使用视图时替换视图，也就是说，不必将定义存储在元数据中。

- 启用按从标量嵌套 select 语句派生的列进行分组，或者按不确定性函数或有外部访问的函数进行分组。
- 在同一语句中多次引用生成的表。

使用 CTE 可以获得提高可读性和轻松维护复杂查询的优点。查询可以分为单独块、简单块、逻辑生成块。之后，这些简单块可用于生成更复杂的临时 CTE，直到生成最终结果集。

可以在用户定义的例程（如函数、存储过程、触发器或视图）中定义 CTE。CTE 由表示 CTE 的表达式名称、可选列列表和定义 CET 的查询组成。定义 CTE 后，可以在 SELECT、INSERT、UPDATE 或 DELETE 语句中对其进行引用，就像引用表或视图一样。CTE 也可用于 CREATEVIEW 语句，作为定义 SELECT 语句的一部分。CTE 的基本语法结构如下：

```
WITH expression_name [(column_name[,...n])]
AS (CTE_query_definition)
```

注意：只有在查询定义中为所有结果列都提供了不同的名称时，列名称列表才是可选的。

运行 CTE 的语句为：

```
SELECT *
FROM cte_name
```

CTE 可以与 ROW_NUMBER()函数结合用于数据库分页。例如要查询客户数据，由于客户数据的数据量较大，如果全部返回，将造成系统载入缓慢，所以对于查询结果数据量较大的情况，一般使用数据库分页，输入需要查询的页数和行数便可只返回指定行数的结果。以 AdventureWorks 2012 数据库为例，分页查询客户数据，获得第 51～60 行的数据的脚本如代码 13.18 所示。

代码 13.18　使用 CTE 和 ROW_NUMBER()进行数据库分页

```
WITH c AS   --定义 CTE
(
SELECT *,ROW_NUMBER() OVER (ORDER BY BusinessEntityID) AS RowID
FROM Person.Person
WHERE EmailPromotion=1
)
SELECT *
FROM c  --使用 CTE
WHERE RowID>50 AND RowID<=60
```

CTE 一个重要的特性就是能够引用其自身，从而创建递归 CTE。递归 CTE 是一个重复执行初始 CTE 以返回数据子集，直到获取完整结果集的公用表表达式。

T-SQL 中的递归 CTE 的结构与其他编程语言中的递归例程相似。但是其他语言中的递归例程返回标量值，递归 CTE 却可以返回多行。递归 CTE 由下列 3 个元素组成。

- 例程的调用。递归 CTE 的第一个调用包括一个或多个由 UNION ALL、UNION、EXCEPT 或 INTERSECT 运算符连接的 CTE_query_definitions。由于这些查询定义形成了 CTE 结构的基准结果集，所以它们被称为“定位点成员”。

CTE_query_definitions 被视为定位点成员，除非它们引用了 CTE 本身。所有定位点成员查询定义必须放置在第一个递归成员定义之前，而且必须使用 UNION ALL 运算符连接

最后一个定位点成员和第一个递归成员。

- 例程的递归调用。递归调用包括一个或多个由引用 CTE 本身的 UNION ALL 运算符连接的 CTE_query_definitions。这些查询定义被称为“递归成员”。
- 终止检查。终止检查是隐式的，当上一个调用中未返回行时，递归将停止。

注意：如果递归 CTE 组合不正确，可能会导致无限循环。

递归查询主要用于树状结构的数据中，即实体对自身是一对多关系，例如部门表，一级部门下有多个二级部门，二级部门下还有多个三级部门……再如商品分类表，大分类下面又分成了多个小分类，小分类下还可以分成多个小分类。

以 AdventureWorks 2012 数据库中的员工为例，要获得经理及向经理报告的雇员的层次列表的脚本，如代码 13.19 所示。

代码 13.19　使用 CTE 递归查询

```
CREATE TABLE dbo.MyEmployees    --为了创建该示例创建的表
(
EmployeeID smallint NOT NULL,
FirstName nvarchar(30)  NOT NULL,
LastName  nvarchar(40) NOT NULL,
Title nvarchar(50) NOT NULL,
DeptID smallint NOT NULL,
ManagerID int NULL,
 CONSTRAINT PK_EmployeeID PRIMARY KEY CLUSTERED (EmployeeID ASC)
);
USE AdventureWorks2012;
GO
WITH DirectReports(ManagerID, EmployeeID, Title, EmployeeLevel) AS
(
    SELECT ManagerID, EmployeeID, Title, 0 AS EmployeeLevel
    FROM dbo.MyEmployees
    WHERE ManagerID IS NULL
    UNION ALL
    SELECT e.ManagerID, e.EmployeeID, e.Title, EmployeeLevel + 1
    FROM dbo.MyEmployees AS e
        INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
SELECT ManagerID, EmployeeID, Title, EmployeeLevel
FROM DirectReports
ORDER BY ManagerID;
GO
```

在 SQL Server 的早期版本中，递归查询通常需要使用临时表、游标和逻辑来控制递归步骤流。递归 CTE 可以极大地简化在 SELECT、INSERT、UPDATE、DELETE 或 CREATEVIEW 语句中运行递归查询所需的代码。

13.1.7　TOP 增强

TOP 子句指定查询结果中只返回第一组行。这组行可以是某一数量的行，也可以是某一百分比数量的行。TOP 表达式可用在 SELECT、INSERT、UPDATE 和 DELETE 语句中。

TOP 子句的语法格式为：

```
TOP (expression) [PERCENT]
    [ WITH TIES ]
```

其中：

- expression 指定返回行数的数值表达式。如果指定了 PERCENT，则 expression 将隐式转换为 float 值；否则，它将转换为 bigint。在 INSERT、UPDATE 和 DELETE 语句中，需要使用括号来分隔 TOP 中的 expression。为保证向后兼容性，支持在 SELECT 中使用不包含括号的 TOP expression，但不推荐这种用法。如果查询包含 ORDER BY 子句，则将返回按 ORDER BY 子句排序的前 expression 行或 expression%的行。如果查询没有 ORDER BY 子句，则行的顺序是随意的。
- PERCENT 指示查询只返回结果集中前 expression%的行。
- WITH TIES 指定从基本结果集中返回额外的行，对于 ORDER BY 列中指定的排序方式参数，这些额外返回行的参数值与 TOP n(PERCENT)行中的最后一行的参数值相同。只能在 SELECT 语句中且只有在指定 ORDER BY 子句之后，才能指定 TOP…WITH TIES。TOP 子句可以支持变量，通过变量指定返回的行数，如代码 13.20 所示为使用变量的 TOP 查询语句。

代码 13.20　使用变量的 TOP 查询

```
USE AdventureWorks ;
GO
DECLARE @p AS int
SET @p=10
SELECT TOP(@p) *          --使用变量作 TOP 查询
FROM HumanResources.Employee;
```

TOP 子句可以用于部分更新或删除数据。例如要删除 1 行学生表中 1 班学生的数据，则对应的脚本如代码 13.21 所示。

代码 13.21　TOP 与 DELETE 命令

```
DELETE TOP(1)   --只删除一行数据
FROM dbo.Student
WHERE ClassID=1
```

如要将 2 班的学生一半转移到 1 班，则对应的 TOP 子句与 UPDATE 语句结合的脚本如代码 13.22 所示。

代码 13.22　TOP 与 UPDATE 命令

```
UPDATE TOP(50) PERCENT --更新一半的数据
dbo.Student
SET ClassID=1
WHERE ClassID=2
```

13.1.8　TABLESAMPLE 子句

TABLESAMPLE 子句将从 FROM 子句中的表返回的行数，限制到样本数或行数的某

一百分比。不能将 TABLESAMPLE 应用于派生表、链接服务器中的表，以及通过表值函数、行集函数或 OPENXML 派生的表。不能在视图或内联表值函数的定义中指定 TABLESAMPLE。

TABLESAMPLE 子句的语法如下：

```
TABLESAMPLE[SYSTEM](sample number[PERCENT|ROWS])
[REPEATABLE(repeat_seed)]
```

当下列任一条件为真时，可以使用 TABLESAMPLE 从大型表中快速返回样本。

- 样本不必是单个行级别的真正随机抽样。
- 该表各页上的行不必与同一页上的其他行相关联。

如果确实需要单个行的随机抽样，则应修改查询以随机筛选出行，而不是使用 TABLESAMPLE。

SYSTEM 指定与 ANSISQL 实现相关的抽样方法。指定 SYSTEM 是可选的，但是此选项是 SQL Server 中唯一可用的抽样方法，并且是默认应用的方法。

TABLESAMPLE SYSTEM 返回行的近似百分比，并针对表中每个物理 8KB 页生成随机值。样本中包括或不包括页均可，具体情况根据页的随机值及查询中指定的百分比来确定。样本中包含的每一页都返回样本结果集中的所有行。例如，如果指定 TABLESAMPLE SYSTEM 10 PERCENT，则 SQL Server 返回该表的大约 10%的指定数据页中的所有行。如果这些行均匀分布在表的各页上，并且表中存在足够多的页，则返回的行数应接近所需样本的大小。但是，由于针对每页生成的随机值与针对任何其他页生成的值无关，因此，返回的页百分比可能会大于或小于所需的百分比。可以使用 TOP(n)运算符将行数限制到指定的最大数。

指定行数，而不是指定基于表中总行数的百分比时，此数将转换为行数的百分比，进而转换为应返回的页数的百分比。然后使用此计算得到的百分比执行 TABLESAMPLE 操作。

如果表由一页组成，则返回此页上的所有行或不返回任何行。在这种情况下，TABLESAMPLE SYSTEM 只能返回页上的 100%或 0%行，而无论页上的行数是多少。

针对特定表使用 TABLESAMPLE SYSTEM，限制了在此表中使用表扫描计划的执行（如果存在堆或聚集索引，则为堆的扫描或聚集索引的扫描）。尽管计划显示已执行表扫描，但实际上只需要从数据文件中读取结果集中包含的那些页。

注意： 使用 TABLESAMPLE SYSTEM 子句时应谨慎，还应了解使用样本的某些影响。例如，两个表的连接可能返回两个表中每行的匹配行。但是，如果对两表中任意一个指定 TABLESAMPLE SYSTEM，则从未抽样表中返回的某些行不太可能具有抽样表中的匹配行。此行为可能使程序员怀疑基础表中是否存在数据一致性的问题，但实际上数据是有效的。同样，如果针对连接的两个表指定 TABLESAMPLE SYSTEM，则发现的问题可能会更严重。

使用 REPEATABLE 选项会导致再次返回选定的样本。使用同一个 repeat_seed 值指定 REPEATABLE 时，只要未对表进行任何更改，SQL Server 将返回相同的行子集。使用其他 repeat_seed 值指定 REPEATABLE 时，SQL Server 通常将返回表中行的不同样本。对表

执行以下操作将视为更改：插入、更新、删除、索引重建、索引碎片整理、还原数据库和附加数据库。

例如要查询 Person.BusinessEntityContact 表，获得其中大约 10%的数据，则使用 TABLESAMPLE 的脚本如代码 13.23 所示。

代码 13.23　使用 TABLESAMPLE

```
USE AdventureWorks 2012;
GO
SELECT *
FROM Person.BusinessEntityContact
TABLESAMPLE (10 PERCENT) ; --获取 10%的数据
```

若要获得大约 200 行数据，则对应的脚本如代码 13.24 所示。

代码 13.24　使用指定行数的 TABLESAMPLE

```
USE AdventureWorks 2012;
GO
SELECT *
FROM Person.BusinessEntityContact
TABLESAMPLE (200 ROWS) ;   --获取大约 200 行数据
```

注意：虽然指定了 200 行数据，但是系统返回的行数却是个随机值，可能返回 80 行数据，也可能返回 400 行数据，如果在 TABLESAMPLE 指定的数值较小，比如 5、8 之类的，系统可能不返回任何结果。

13.2　SQL Server 2008 新增语法

SQL Server 2008 在保留了原有版本语法的基础上对 T-SQL 语句进行了进一步的增强。新的数据类型将在其他章节进行讲解，本节主要讲解在 SQL Server 2008 中新增加的语法。

13.2.1　T-SQL 基础增强

SQL Server 2008 中将 T-SQL 语句向 C 语言格式靠拢，可以像 C 语言那样定义和赋值同时进行，可以使用累加运算符等。在老版本的 SQL 中，一个变量的定义和赋值是分开的，例如：

```
DECLARE @i int
SET @i=2
```

这种方式显得十分冗余，SQL Server 2008 中可以直接将变量的定义和赋值写在一句中完成。上面声明的变量可以改写为：

```
DECLARE @i int=2
```

在 C 语言中经常使用累加运算符进行运算，使得代码更加简洁，常用的累加运算符包括：+=、–=、*=、/=、|=、&=、^=、%=。现在在 SQL Server 2008 中也可以使用累加运算

符。例如声明一个变量，然后将该变量乘 2，则对应的脚本如代码 13.25 所示。

代码 13.25　累加运算符

```
DECLARE @i int=2     --定义变量并初始化值
SET @i*=2            --累加运算符
PRINT @i
```

其他运算符的使用方法类似。

SQL Server 2008 支持在一条 INSERT 命令中一次插入多个数据记录。只需要在 VALUES 关键字后跟多条记录，每条记录以逗号分隔即可。例如创建一个临时表，一次插入 2 条记录，然后再与另外 2 条记录连接的脚本如代码 13.26 所示。

代码 13.26　一次插入多条记录

```
CREATE TABLE #t(c1 int,c2 nvarchar(10))
INSERT #t VALUES(1,'a'),(2,'b')     --一次插入多条数据

SELECT c1,c2,c3
FROM #t
INNER JOIN
(
  VALUES (1, 'aa'), (3, 'cc')        --内连接多条数据
) t (c3, c4)
ON #t.c1 = t.c3;
```

13.2.2　Grouping Sets 语法

使用 GROUPING SETS 的 GROUP BY 子句，可以生成一个等效于由多个简单 GROUP BY 子句的 UNION ALL 生成的结果集。GROUPING SETS 可以生成等效于由简单 GROUP BY、ROLLUP 或 CUBE 操作生成的结果。

例如，根据 PersonType 和 EmailPromotion 对 Person.Person 表进行分组，其对应的脚本如代码 13.27 所示。

代码 13.27　GROUPING SETS 的使用

```
SELECT PersonType,EmailPromotion,Count(1)
FROM Person.Person
GROUP BY GROUPING SETS (PersonType,EmailPromotion)    --使用 PersonType 和
                                                      EmailPromotion 分组
```

其结果与代码 13.28 结果相同，这两个语句是等价的。

代码 13.28　GROUPING SETS 的等价语句

```
SELECT NULL AS PersonType,EmailPromotion,Count(1)
FROM Person.Person
GROUP BY EmailPromotion--使用 EmailPromotion 进行分组
UNION ALL
SELECT PersonType,NULL AS EmailPromotion,Count(1)
FROM Person.Person
GROUP BY PersonType--使用 PersonType 进行分组
GROUP BY ContactID  --使用 ContractID 进行分组
```

GROUPING SETS、ROLLUP 或 CUBE 的不同组合可以生成等效的结果集。GROUPING SETS 与其等价表达式如表 13.5 所示。

表 13.5　GROUPING SETS的等价表达式

GROUPING SETS 表达式	等价表达式
GROUP BY GROUPING SETS (())	GROUP BY ()
GROUP BY GROUPING SETS ((C1, C2, ..., Cn))	GROUP BY C1, C2, ..., Cn
GROUP BY GROUPING SETS ((C1, C2, …, Cn–1, Cn) ,(C1, C2, ..., Cn-1) ... ,(C1, C2) ,(C1) ,())	GROUP BY ROLLUP(C1, C2, …, Cn–1, Cn)
GROUP BY GROUPING SETS ((C1, C2, C3, ..., Cn–2, Cn–1, Cn) -- All dimensions are included. ,(, C2, C3, ..., Cn–2, Cn–1, Cn) -- n–1 dimensions are included. ,(C1, C3, ..., Cn–2, Cn–1, Cn) … ,(C1, C2, C3, ..., Cn–2, Cn–1,) ,(C3, ..., Cn-2, Cn–1, Cn) -- n–2 dimensions included ,(C1 ..., Cn-2, Cn–1, Cn) … ,(C1, C2) -- 2 dimensions are included ,… ,(C1, Cn) ,… ,(Cn–1, Cn) ,… ,(C1) -- 1 dimension included ,(C2) ,… ,(Cn–1) ,(Cn) ,()) -- Grand total, 0 dimension is included	GROUP BY CUBE (C1, C2, C3, ..., Cn-2, Cn-1, Cn)
GROUP BY GROUPING SETS ((C1, C2, C3) ,(C1, C2) ,(C1, C3) ,(C2, C3) ,(C1) ,(C2) ,(C3) ,())	GROUP BY CUBE (C1, C2, C3)
GROUPING SETS ((A, C1, C2, ..., Cn), (A), ())	ROLLUP(A, (C1, C2, ..., Cn))
GROUPING SETS ((), (A), (C1, C2, ..., Cn), (A, C1, C2, ..., Cn))	CUBE(A, (C1, C2, ..., Cn))

续表

GROUPING SETS 表达式	等价表达式
GROUP BY GROUPING SETS ((A), (A, B), (A, C), (A, B, C))	GROUP BY A, CUBE (B, C)
GROUP BY GROUPING SETS ((A, B), (A, C))	GROUP BY A, GROUPING SETS ((B), (C))
GROUP BY GROUPING SETS ((),(C),(C,D),(A),(A,C),(A,C,D),(A,B),(A,B,C),(A,B,C,D))	GROUP BY ROLLUP (A, B), ROLLUP(C, D)
GROUP BY GROUPING SETS ((A), ROLLUP (B, C))	GROUP BY GROUPING SETS ((A), (B,C), (B), ())
GROUP BY GROUPING SETS(A, (B, ROLLUP(C, D)))	GROUP BY GROUPING SETS (A, B, (B,C), (B, C, D) ())

13.2.3　Merge 语法

在 SQL Server 2008 中，可以使用 MERGE 在一条语句中执行 INSERT、UPDATE 和 DELETE 操作。MERGE 语法允许将数据源与目标表或视图连接，然后根据该连接的结果执行多项操作。MERGE 语法包括如下四个主要子句。

- MERGE 子句用于指定作为 INSERT、UPDATE 或 DELETE 操作目标的表或视图。
- USING 子句用于指定要与目标连接的数据源。
- ON 子句用于指定决定目标与源表的匹配位置的连接条件。
- WHEN 子句用于根据 ON 子句的结果指定要执行的操作。

Merge 语句的语法格式如代码 13.29 所示。

代码 13.29　Merge 语法

```
MERGE
        [ TOP ( expression ) [ PERCENT ] ]
        [ INTO ] target table [ WITH ( <merge hint> ) ] [ [ AS ] table alias ]
        USING <table source>
        ON <merge search condition>
        [ WHEN MATCHED [ AND <clause_search_condition> ]
            THEN <merge_matched> ]
        [ WHEN NOT MATCHED [ BY TARGET ] [ AND <clause_search_condition> ]
            THEN <merge_not_matched> ]
        [ WHEN NOT MATCHED BY SOURCE [ AND <clause_search_condition> ]
            THEN <merge_matched> ]
        [ <output_clause> ]
        [ OPTION ( <query_hint> [ ,...n ] ) ]
```

其中几个主要的参数是：

- target_table 是目标表的表名。
- table_source 是进行合并的源表表名。
- ON <merge_search_condition>用于指定源表到目标表合并时使用的比对项，一般情况下这里是指定两个表的主键。
- WHEN MATCHED THEN <merge_matched>表示通过前面的 ON < merge_search_condition >，找到了匹配项时应该执行的操作。

- WHEN NOT MATCHED [BY TARGET] THEN <merge_not_matched>表示通过合并比对，在目标表中没有找到源表中的匹配数据时执行的操作。
- WHEN NOT MATCHED BY SOURCE THEN <merge_matched>表示通过合并比对，在源表中没有找到目标表中的匹配数据时执行的操作。
- <output_clause>返回在对目标表执行了 INSERT、UPDATE 和 DELTE 操作的数据。在输出结果中有一个特殊的列$action，该列是 INSERT、UPDATE 或 DELETE 中的一个值，用于表示当前行执行的操作。

例如创建一个源表并初始化数据，如代码 13.30 所示。

代码 13.30　创建并初始化源表

```
CREATE TABLE tbSource   --创建源表和数据
(
    C1 INT PRIMARY KEY,
    C2 NVARCHAR(10)
)
GO
INSERT tbSource
VALUES(1,N'甲'),
(2,N'乙')
GO
```

接下来根据源表创建目标表，并且对源表的数据进行更改，如代码 13.31 所示。

代码 13.31　创建目标表和修改源表数据

```
SELECT * INTO tbTarget  --创建目标表
FROM tbSource
--以下是对源表数据的修改
DELETE tbSource
WHERE C1=1
UPDATE tbSource
SET C2=N'乙'
WHERE C1=2
INSERT tbSource
VALUES(3,N'丙')
```

现在源表和目标表的数据如表 13.6 所示。

表 13.6　源表和目标表的数据

tbSource		tbTarget	
C1	C2	C1	C2
2	乙 2	1	甲
3	丙	2	乙

下面需要将源表的数据合并到目标表中，数据合并主要包括以下 3 个操作。

- 源表删除的数据，目标表也应该被删除。
- 源表修改的数据，目标表也进行修改。
- 源表增加的数据也需要插入到目标表中。

整个合并操作以两个表的主键 C1 相同为依据，使用 Merge 语句进行合并的脚本如代

码 13.32 所示。

代码 13.32　使用 Merge 合并表

```
MERGE tbTarget t    --合并源表和目标表
  USING tbSource s
  ON t.c1 = S.c1    --合并依据
  WHEN MATCHED THEN --修改
    UPDATE SET t.c2 = S.c2
  WHEN NOT MATCHED THEN                   --新增
    INSERT VALUES(c1, c2)
  WHEN NOT MATCHED BY SOURCE THEN         --删除
    DELETE
  OUTPUT $action, INSERTED.c1 [New c1],--输出合并操作结果
                  INSERTED.c2 [New c2],
                  DELETED.c1  [Original c1],
                  DELETED.c2  [Original c2];
```

系统输出结果如表 13.7 所示。

表 13.7　Merge输出结果

$action	New c1	New c2	Original c1	Original c2
DELETE	NULL	NULL	1	甲
UPDATE	2	乙 2	2	乙
INSERT	3	丙	NULL	NULL

从 Merge 的输出可以看出，目标表中的“1，甲”数据行被删除了，“2，乙”数据行被修改为“2，乙 2”，还插入了一行数据“3，丙”。接下来再查询目标表，可以看到运行 Merge 语句以后，目标表的数据已经和源表的数据相同了。

13.2.4　表值参数 TVP

表值参数（Table-valued Parameters，TVP）是 SQL Server 2008 中的新参数类型。表值参数是使用用户定义的表类型来声明的。使用表值参数，可以不必创建临时表或许多参数，即可向 Transact-SQL 语句或例程（如存储过程或函数）发送多行数据。表值参数与 OLEDB 和 ODBC 中的参数数组类似，但具有更高的灵活性，且与 Transact-SQL 的集成更紧密。表值参数的另一个优势是能够参与基于数据集的操作。

注意：Transact-SQL 通过引用向例程传递表值参数，以避免创建输入数据的副本。

可以使用表值参数创建和执行 Transact-SQL 例程，并且可以使用任何托管语言从 Transact-SQL 代码、托管客户端及本机客户端调用它们。

表值参数具有两个主要部分：SQL Server 类型及引用该类型的参数。若要创建和使用表值参数，可执行以下步骤：

（1）创建表类型并定义表结构。

（2）声明具有表类型参数的例程（存储过程或函数）。

（3）声明表类型变量，并引用该表类型。

（4）使用 INSERT 语句填充表变量。

（5）创建并填充表变量后，可以将该变量传递给例程。

例程超出作用域后，表值参数将不再可用。类型定义则会一直保留，直到被删除。

表值参数具有更高的灵活性，在某些情况下，使用该参数可比使用临时表或其他传递参数列表能提供更好的性能。表值参数具有以下优势：

- 首次从客户端填充数据时，不获取锁。
- 提供简单的编程模型。
- 允许在单个例程中包括复杂的业务逻辑。
- 减少到服务器的往返。
- 可以具有不同基数的表结构。
- 是强类型。
- 使客户端可以指定排序顺序和唯一键。

表值参数有下面的限制：

- SQL Server 不维护表值参数列的统计信息。
- 表值参数必须作为输入 READONLY 参数传递到 Transact-SQL 例程。不能在例程体中对表值参数执行诸如 UPDATE、DELETE 或 INSERT 这样的 DML 操作。
- 不能将表值参数用做 SELECTINTO 或 INSERT EXEC 语句的目标。表值参数可以用在 SELECT INTO 的 FROM 子句中，也可以用在 INSERT EXEC 字符串或存储过程中。

就像其他参数一样，表值参数的作用域也是存储过程、函数或动态 Transact-SQL 文本。同样，表类型变量也与使用 DECLARE 语句创建的其他任何局部变量一样具有作用域。可以在动态 Transact-SQL 语句内声明表值变量，并且可以将这些变量作为表值参数传递到存储过程和函数。

例如对于前面提到的学生成绩表 Student，可以创建一个表值类型用于表示该对象，具体创建脚本如代码 13.33 所示。

代码 13.33　创建表值类型

```
CREATE TYPE StudentTableType AS TABLE      --创建表值类型
(
    StudentID int,
    ClassID int,
    Mark int
)
```

接下来创建存储过程用于添加 Student 记录，该存储过程即可使用表值参数，如代码 13.34 所示。

代码 13.34　使用表值参数的存储过程

```
CREATE PROC AddStudent
@stuTable StudentTableType READONLY    --表值参数作为存储过程的参数
AS
INSERT INTO Student
SELECT *
FROM @stuTable                         --使用传入的表值参数
```

注意：在使用表值参数时，必须声明表值参数为 READONLY，否则创建存储过程将失败。

创建存储过程后即可在外部调用该存储过程，例如使用表值参数添加 3 个学生成绩数据如代码 13.35 所示。

代码 13.35　调用有表值参数的存储过程

```
DECLARE @stuTable StudentTableType --定义表值变量
INSERT INTO @stuTable                --初始化表值变量内容
VALUES (20,1,95),(21,1,78),(22,2,88)
--调用有表值参数的存储过程
EXEC AddStudent @stuTable
```

系统提示 3 行数据受影响，查询 Student 表将看到数据已经成功插入。

13.3　SQL Server 2012 新增语法

SQL Server 2012 是在 SQL Server 2008 的基础上对基本语句有所增强，同时也有一些特性的增加，在前面的第 4 章中已经对一些新特性有所讲解。本章就将详细介绍新增加的语法。

13.3.1　Execute 语法

Execute 是用于执行系统存储过程、用户定义存储过程以及函数的语句。SQL Server 2012 中添加了 WITH RESULT SETS 选项，可以用于更改存储过程返回的结果集中的列名和数据类型。WITH RESULT SETS 选项既可以应用在单一结果集的结果中，也可以应用在多个结果集的结果中。

在数据库 AdventureWorks2012 中存在一个名为 uspGetEmployeeManagers 的存储过程，并且该存储过程需要一个参数员工编号（EmployeeID）。为了能够让读者了解 WITH RESULT SETS 选项的作用，现在先将员工编号为 2 的员工信息，通过 uspGetEmployeeManagers 存储过程查询出来，具体脚本如代码 13.36 所示。

代码 13.36　调用存储过程 uspGetEmployeeManagers

```
USE AdventureWorks2012;
EXEC uspGetEmployeeManagers 2
```

执行效果如图 13.1 所示。

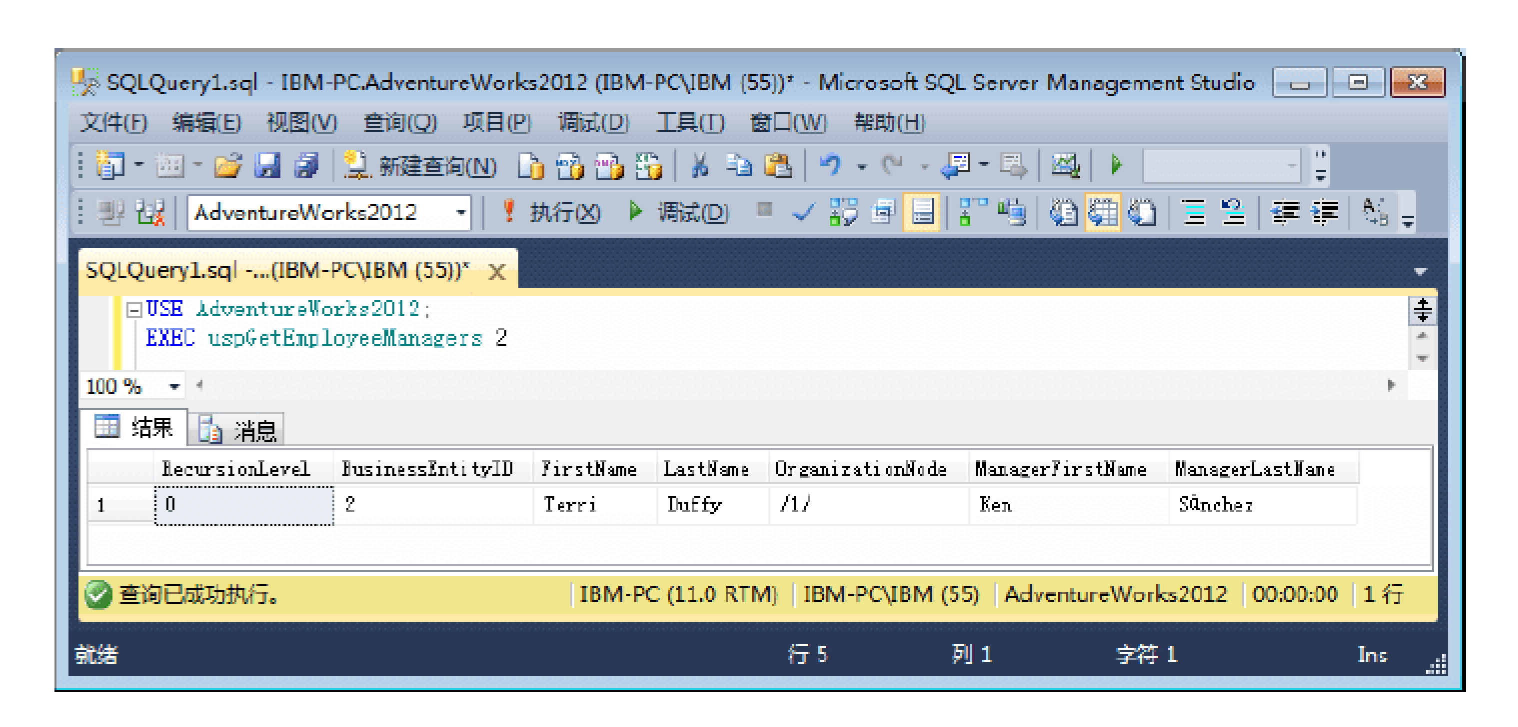


图 13.1　执行存储过程 uspGetEmployeeManagers

如果在执行上述存储过程时，加上 WITH RESULT SETS 选项，并重新定义结果集中的列名和数据类型。为了能够使结果的变化明显，这里将所有的列名都改成中文，脚本如代码 13.37 所示。

代码 13.37　调用存储过程 uspGetEmployeeManagers 并加上 WITH RESULT SETS 选项

```
USE AdventureWorks2012;
EXEC uspGetEmployeeManagers 2
WITH RESULT SETS
(
   ([级别] int NOT NULL,
    [编号] int NOT NULL,
    [员工的名] nvarchar(50) NOT NULL,
    [员工的姓氏] nvarchar(50) NOT NULL,
    [员工的经理编号] nvarchar(max) NOT NULL,
    [经理的名] nvarchar(50) NOT NULL,
    [经理的姓氏] nvarchar(50) NOT NULL
    )
);
```

执行效果如图 13.2 所示。

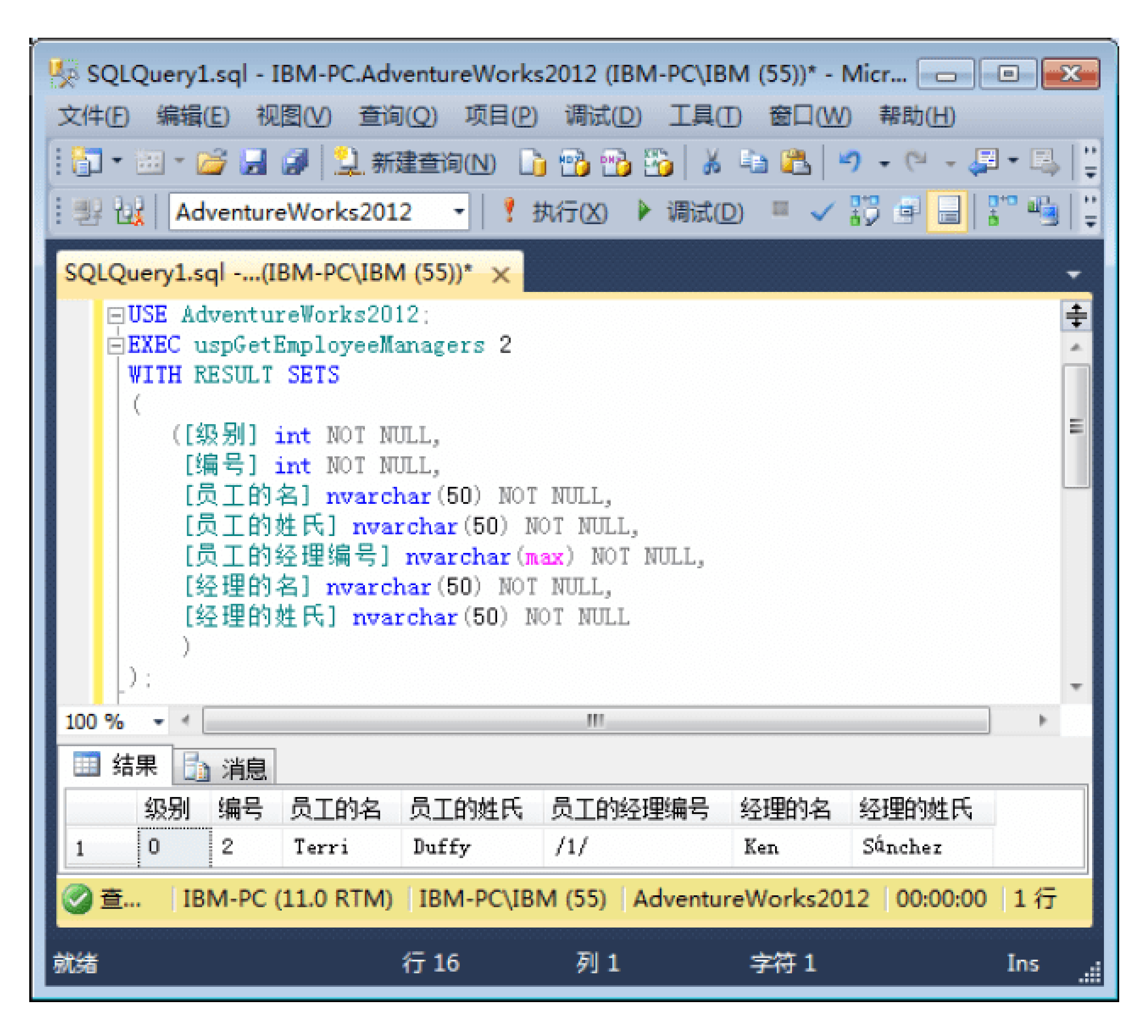


图 13.2　带 WITH RESULT SETS 选项执行存储过程 uspGetEmployeeManagers

注意：WITH RESULT SETS 选项后面列的数量要与存储过程中查询出的列数量一致，否则就会出现错误。

WITH RESULT SETS 选项除了可以定义单一结果集外，还可以重新定义多个结果集，下面就先建立一个返回 2 个结果集的存储过程，脚本如代码 13.38 所示。

代码 13.38　创建返回两个结果集的存储过程

```
USE AdventureWorks2012;
CREATE PROC test AS
-- 第 1 个结果集
```

```
SELECT ProductID, Name, ListPrice
FROM Production.Product
-- 第 2 个结果集
SELECT LocationID,Name
FROM Production.Location
```

使用 WITH RESULT SETS 选项在执行上面的存储过程时重新定义这个结果集，脚本如代码 13.39 所示。

代码 13.39　创建返回 2 个结果集的存储过程

```
USE AdventureWorks2012;
EXEC test
WITH RESULT SETS
(
   ([产品编号] int,
    [产品名称] nvarchar(50),
    [价格] money)
    ,
    ([编号] int,
     [名称] nvarchar(50))
);
```

执行效果如图 13.3 所示。

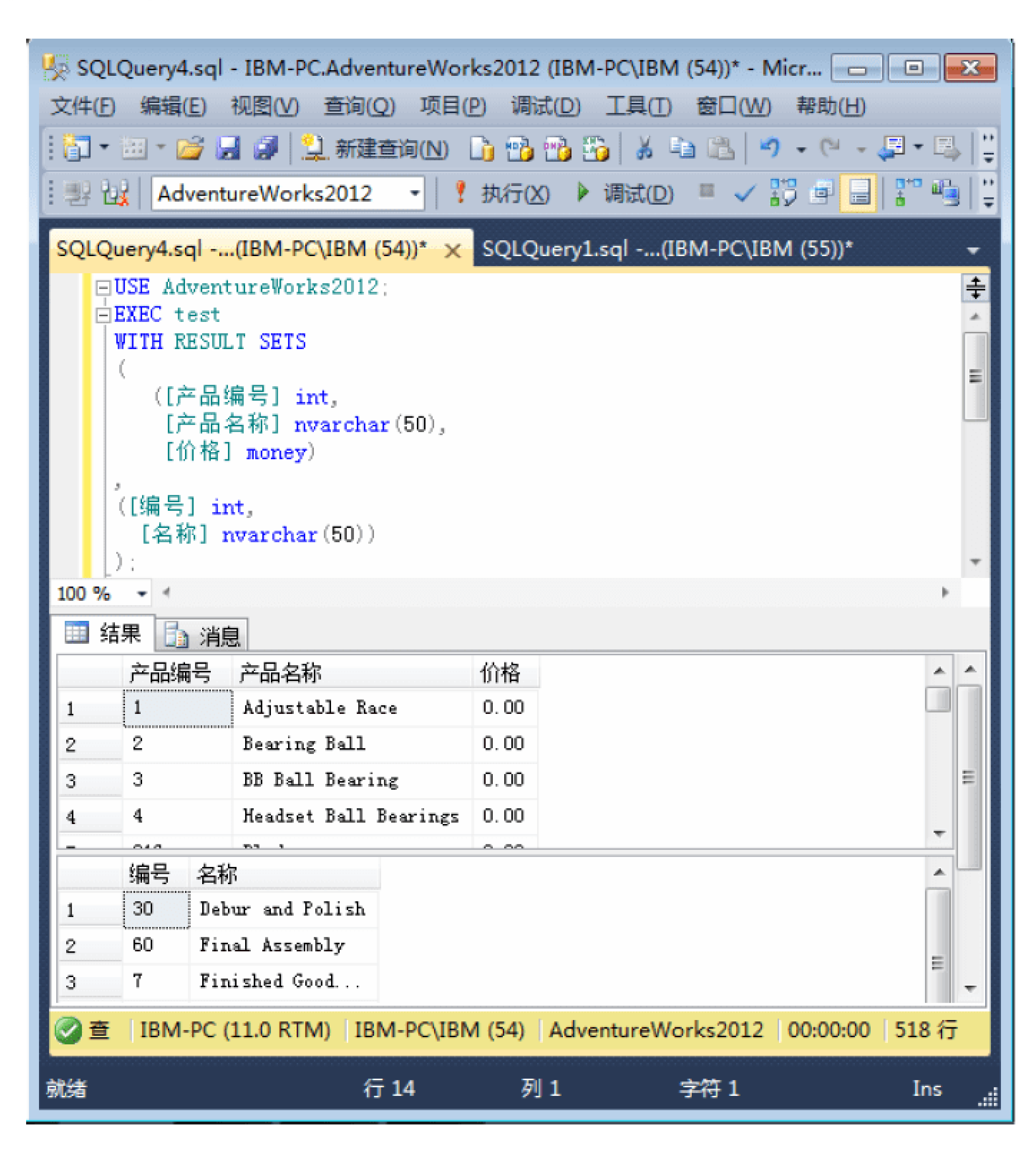


图 13.3　使用 WITH RESULT SETS 重新定义两个结果集

说明： 使用 WITH RESULT SETS 重新定义多个结果集时，每个结果集之间要用逗号隔开，并且每个结果集中的列数目也应该与存储过程中原结果集的数目相同。

13.3.2　实现即席查询分页 Order by

Order by 子句主要用于在查询时对查询结果进行排序，在 SQL Server 2012 中，Order by 子句新增了分页的功能。所谓分页的功能，就是能够将结果中的数据按照要求去除一定数量的数据。比如：从查询结果中取出从第 5 个记录开始后的 10 条记录等。Order by 子句的分页功能主要是通过 OFFSET 和 FETCH 关键字实现的，具体的语法规则如下所示。

```
SELECT * FROM tables
ORDER BY column1
OFFSET rowscount1 ROWS
FETCH NEXT rowscount2 ROWS ONLY
```

其中：

- column1：列名，按照该列对数据表中的数据排序，也可以指定多个列同时排序，多个列之间用逗号隔开即可。
- rowscount1：指定查询结果中要忽略的行数，也就是要跳过的行数。
- rowscount2：指定查询结果中要返回的行数。

按照上面的语法规则，从 AdventureWorks2012 中的 Production.Product 表中查询数据，要求返回该表中从第 2 行开始的 5 行数据。具体的脚本如代码 13.40 所示。

代码 13.40　使用 Order by 子句查询指定行数的数据

```
USE AdventureWorks2012;
SELECT * FROM Production.Product
ORDER BY ProductID
OFFSET 1 ROWS
FETCH NEXT 5 ROWS ONLY
```

执行效果如图 13.4 所示。

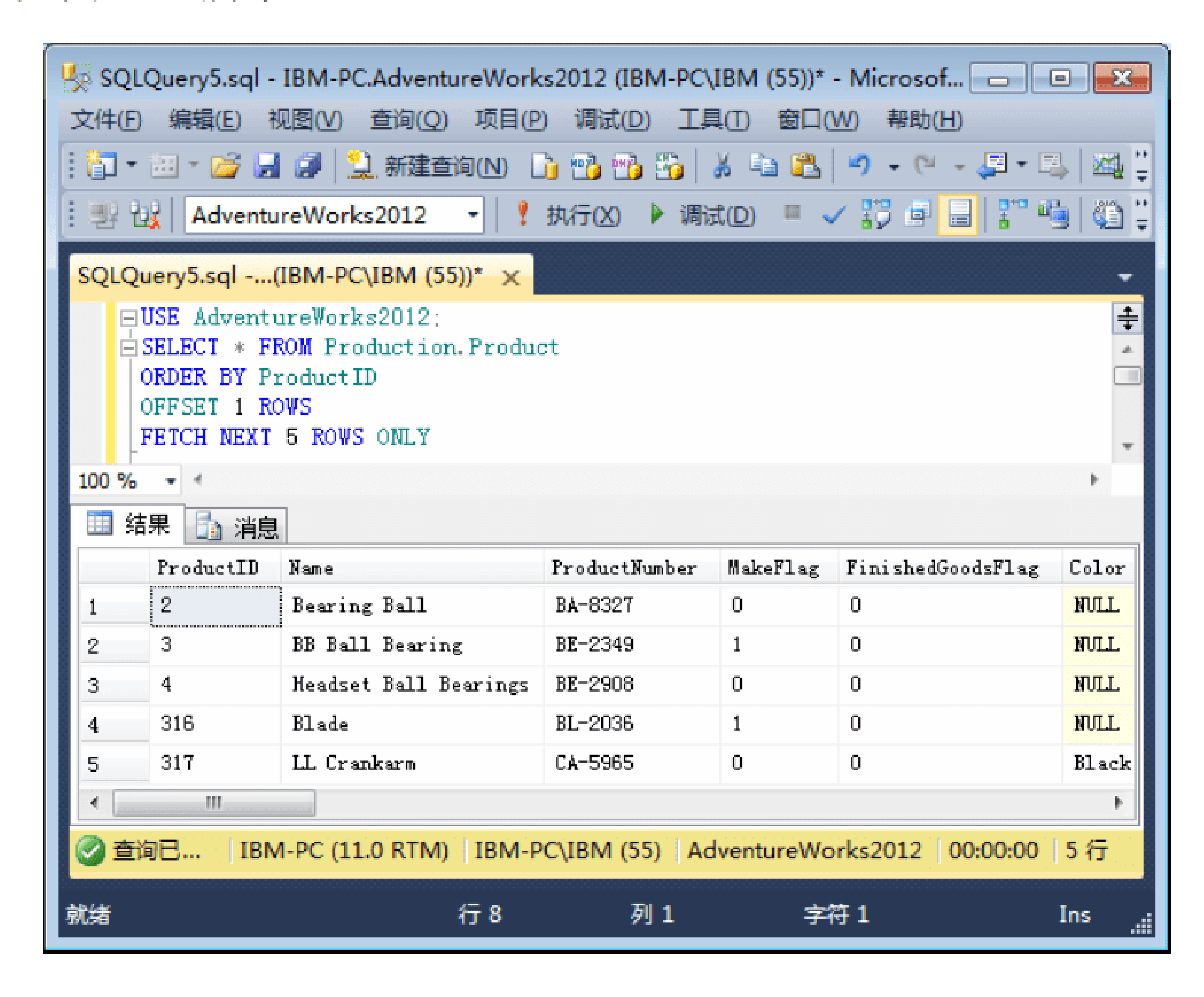


图 13.4　使用 Order by 子句查询指定行数的数据

从查询结果可以看出，结果查询的是从 2 行开始的 5 条数据。这里在 OFFSET 和 FETCH 后面都是用的常数，如果把它们的值都换成变量，那么可以很容易地完成数据分页的操作。

13.3.3　SEQUENCE 序列对象

序列对象是指生成一个数字序列，类似于标识列，但是与标识列不同的是该序列对象指定的数值可以重复使用，比如：为表中的某列重复设置从 1～5 的数值。此外，序列对象还可以使用语句对数值进行重置，比如：序列从 1 增长到了 10，现在还可以通过重置，将序列再次从 1 开始增加。因此，序列与标识列比较而言，序列更占优势。

1．创建序列

创建序列的语法如下所示。

```
CREATE SEQUENCE [schema_name . ] sequence_name
    [ AS [ built_in_integer_type | user-defined_integer_type ] ]
    [ START WITH <constant> ]
    [ INCREMENT BY <constant> ]
    [ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]
    [ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]
    [ CYCLE | { NO CYCLE } ]
    [ ; ]
```

其中：

- ❑ sequence_name：序列的名字，在数据库中序列的名字是唯一的。
- ❑ [built_in_integer_type | user-defined_integer_type]：系统或用户自定义的整数类型，包括 tinyint、smallint、int、bigint、decimal、numeric，其中 decimal、numeric 小数位数要是 0。如果不指定数据类型，将会默认使用 bigint 类型。
- ❑ START WITH：序列返回的第 1 个值，该值必须在序列的最大值和最小值之间。
- ❑ INCREMENT BY：序列中的值每次递增或递减的变化量。如果为负值就为递减量。
- ❑ MINVALUE：指定序列中的最小值，如果不指定最小值，默认的最小值就是指定数据类型的最小值。这里只有 tinyint 的最小值是 0，其余都是负数。
- ❑ MAXVALUE：指定序列中的最大值，如果不指定最大值，默认的最大值就是指定数据类型的最大值。
- ❑ CYCLE | { NO CYCLE }：指定序列中的值是否循环使用，如果是 CYCLE 则是循环使用。

在数据库 TestDb1 中，创建一个用户信息表（userinfo），创建表的脚本如代码 13.41 所示。

代码 13.41　创建表 userinfo

```
USE TestDb1;
CREATE TABLE userinfo
(
   id  int,
   username nvarchar(20),
   userpwd  nvarchar(20)
)
```

为该表创建一个序列，要求从 1 开始增加，增量是 2。具体的脚本如代码 13.42 所示。

代码 13.42　创建序列 userinfo_seq

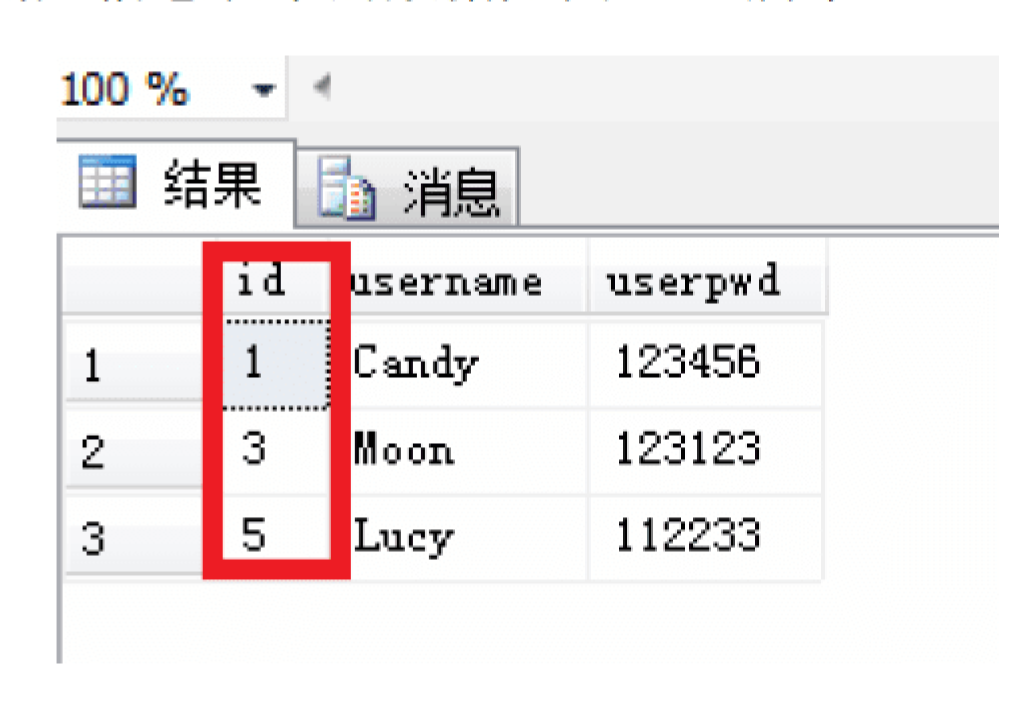
```
USE TestDb1;
CREATE SEQUENCE userinfo_seq
AS int
START WITH 1
INCREMENT BY 2;
```

创建好序列后，就可以使用序列向数据表 userinfo 中添加数据了，这里向 userinfo 表中添加 3 条数据。使用序列添加值要是用 NEXT VALUE FOR 子句完成，具体脚本如代码 13.43 所示。

代码 13.43　使用序列向 userinfo 表中添加值

```
INSERT userinfo
     VALUES (NEXT VALUE FOR userinfo_seq, 'Candy', '123456');
INSERT userinfo
     VALUES (NEXT VALUE FOR userinfo_seq, 'Moon', '123123');
INSERT userinfo
     VALUES (NEXT VALUE FOR userinfo_seq, 'Lucy', '112233');
```

执行上面的语句，用户信息表中的数据如图 13.5 所示。



100 %　结果　消息

	id	username	userpwd
1	1	Candy	123456
2	3	Moon	123123
3	5	Lucy	112233

图 13.5　userinfo 表中的数据

从图 13.5 中的查询效果可以看出，使用序列插入值后 id 列为 1,3,5。

2．修改序列

序列创建完成后，能够修改序列中的最大值、最小值及重新开始序列的值，具体语法如下：

```
ALTER SEQUENCE [schema_name . ] sequence_name
    [ RESTART [ WITH <constant> ] ]
    [ INCREMENT BY <constant> ]
    [ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]
    [ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]
    [ CYCLE | { NO CYCLE } ]
    [ ; ]
```

该语法与创建序列的语法类似，只是多了一个 RESTART 子句，该子句主要用于指定该序列重新开始的数值，该数值必须在最大值和最小值之间。

将前面创建的序列 userinfo_seq 的值修改成从 3 开始，具体的脚本如代码 13.44 所示。

代码 13.44　修改序列 userinfo_seq

```
USE TestDb1;
ALTER SEQUENCE userinfo_seq
RESTART WITH 3;
```

3．删除序列

序列作为数据库中的一个对象，也是使用 DROP 语句删除的，具体语法如下：

```
DROP SEQUENCE sequence_name;
```

这里，sequence_name 就是序列的名称。

将序列 userinfo_seq 删除的脚本如代码 13.45 所示。

代码 13.45　删除序列 userinfo_seq

```
USE TestDb1;
DROP SEQUENCE userinfo_seq;
```

注意：即使使用过该序列添加值，序列仍旧可以删除，并且添加的值也不会被删除。

13.3.4　THROW 语句

从 SQL Server 2005 开始就引入了异常处理机制 TRY…CATCH 子句，在 SQL Server 2012 中又在异常处理中增加了 THROW 子句，使其可以在语句中抛出异常由 catch 子句处理。THROW 子句的语法如下所示。

```
THROW [ { error number | @local variable },
        { message | @local variable },
    { state | @local_variable }
  ];
```

其中：

- error_number：表示异常的常量或变量，错误号是在 50000～2147483647 范围内。
- message：描述异常的字符串。
- state：指示与消息连接的状态。

使用 THROW 抛出错误号是 60000 的异常，具体脚本如代码 13.45 所示。

```
THROW 60000, 'The record error.', 1;
```

此外，THROW 还可以用在 CATCH 子句中，再次抛出异常。

注意：使用 THROW 子句抛出异常时，THROW 前面的语句必须以分号结束。

13.4　小　　结

本章主要讲解了在 SQL Server 2005、SQL Server 2008 及 SQL Server 2012 中新增的 T-SQL 语法。

在 SQL Server 2005 中提供的排名函数如下：

- RANK()；
- DENSE_RANK()；
- NTILE()；
- ROW_NUMBER()。

其中的 ROW_NUMBER()函数和公用表表达式一起使用，常用于数据库分页。

SQL Server 2005 中 TRY 和 CATCH 块用于异常处理，在 CATCH 块中可以通过 ERROR_MESSAGE 等函数获得异常的详细信息。APPLY 操作符用于表值函数和表或视图的连接操作。APPLY 操作分为 CROSS APPLY 和 OUTER APPLY 两种。PIVOT 和 UNPIVOT 运算符主要用于行列转换。OUTPUT 子句则用于输出被插入、删除和修改的数据或者将这些数据转移备份到其他表中。公用表表达式除了用于分页外，主要用于递归查询。TOP 子句可以使用参数指定返回的行数。TABLESAMPLE 子句将从 FROM 子句中的表返回的行数限制到样本数或行数的某一百分比。

在 SQL Server 2008 中对基础 T-SQL 进行了增强，可以同时声明变量和赋值、使用累加运算符和一次插入多条记录。另外提供了 Grouping Sets 子句可以生成等效于由简单 GROUP BY、ROLLUP 或 CUBE 操作生成的结果。Merge 语句可以提供两个表的合并，同时还可以将合并操作输出。在存储过程和函数中可以使用表值参数。

在 SQL Server 2012 中也对 T-SQL 语句有所增强，主要表现在 Execute 语句、Order by 语句中，另外还添加了序列 SEQUENCE 对象及在异常处理中添加了 THROW 语句。其中，SEQUENCE 对象对应于原有的标识列设置显示出更大的优势，以使用户可以更方便地使用序列。

第 14 章　Service Broker——异步应用程序平台

SQL Server Service Broker（SSB）是 SQL Service 2005 中引入的新技术，作为数据库的一部分，用来构建可靠的、可扩展的、异步的、分布式数据库应用程序。本章将主要介绍 Service Broker 的基础知识和使用。

14.1　Service Broker 简介

对于一直使用 SQL Server 2000 的用户来说，Service Broker 是一个全新的名词，Service Broker 也许是 SQL Server 2005 中功能最强大，但最不为用户熟知的新功能。Service Broker 使得 SQL Server 成为一个构建可靠的分布式异步应用程序的平台。

14.1.1　Service Broker 是什么

Service Broker 是在 Microsoft SQL Server 2005 中才开始使用的新技术，它可帮助数据库开发人员生成安全、可靠且可伸缩的应用程序。由于 Service Broker 是数据库引擎的组成部分，因此管理这些应用程序就成为数据库日常管理的一部分。

Service Broker 为 SQL Server 提供队列和可靠的消息传递。Service Broker 对使用单个 SQL Server 实例的应用程序和在多个实例间分配工作的应用程序都适用。

在单个 SQL Server 实例中，Service Broker 提供了可靠的异步编程模型。数据库应用程序通常使用异步编程来缩短交互式响应时间，并增加应用程序总吞吐量。

Service Broker 还会在 SQL Server 实例之间提供可靠的消息传递服务。Service Broker 可帮助开发人员编写与称为服务的独立的、自包含的组件相关的应用程序。需要使用这些服务中所包含的功能的应用程序，可以利用消息与这些服务进行交互。Service Broker 使用 TCP/IP 在实例之间交换消息。Service Broker 包含一些功能，有助于避免通过网络进行未经授权的访问及对通过网络发送的消息进行加密。

使用 Service Broker，内部或外部应用程序可以使用 Transact-SQL 的数据加工语言（DML），发送和接收可靠的、异步的消息。消息发送到队列中，队列可位于与发送者相同的数据库中、同一 SQL Server 实例的其他数据库中，或者是同一台服务器或者远程服务器的其他 SQL Server 实例中。Service Broker 有三种类型的组件：

- 会话组件。会话组、会话和消息构成 Service Broker 应用程序的运行时结构。应用程序将消息作为会话的一部分进行交换。每个会话都是会话组的一部分，一个会话组可以包含多个会话。每个 Service Broker 会话都是一个对话。对话就是只有两

个参与者交换消息的会话。
- 服务定义组件。这些组件是设计时组件，用于指定应用程序所用会话的基础结构。它们定义应用程序的消息类型、会话流和数据库存储。
- 网络和安全组件。这些组件定义了用来在数据库引擎实例之间交换消息的基础结构。为了帮助数据库管理员管理不断变化的环境，Service Broker 允许管理员独立于应用程序代码来配置这些组件。

在 SQL Server 2008 中，除了保留原有 Service Broker 的功能外还做了以下增强。
- 在对象资源管理器中新增了 Service Broker 元素。已经向 SSMS 对象资源管理器中添加了 Service Broker 菜单项；已经向对象资源管理器层次结构中添加了会话优先级。使用现有的 Service Broker 对象，可在单击右键后出现更多菜单项，其中包括“属性”菜单项。
- 新增了 Service Broker 系统监视器计数器。已经向 Broker 统计信息性能对象中添加了 5 个新的计数器，分别是 Activation Errors Total、Corrupted Messages Total、Dequeued TransmissionQ Msgs/sec、Dropped Messages Total 和 Enqueued TransmissionQ Msgs/sec。
- 新增了会话优先级。管理员和开发人员可以通过会话优先级来指定，首要 Service Broker 会话的消息先于次要会话的消息发送和接收。
- 新增了配置诊断实用工具。ssbdiagnose 实用工具可对两个 Service Broker 服务之间的配置或者单个服务的配置进行分析。检查出的错误在命令提示符窗口以可读文本形式报告，或者以可重定向到文件或其他程序的格式化 XML 形式报告。
- 新增了系统监视器对象。Broker TO 统计信息性能对象可对 Service Broker 诊断请求传输对象的频率，以及将不活动的传输对象写入 tempdb 内工作表的频率进行报告。

在 SQL Server 2008 的基础上，SQL Server 2012 中的 Service Broker 功能也有一定的改进。
- 将消息发送到多个目标服务。该功能是通过支持多个会话句柄，扩展了 Transact-SQL 语句的 SEND 语法以启用多播。
- 通过队列公开此消息的排队时间。该功能的实现是在队列中添加了一个新列，用于存储排队时间。
- 禁用有害消息。通过应用程序来定义如何处理有害消息，主要是在使用 CREATE QUEUE 和 ALTER QUEUE 语句时添加子句 POISON_MESSAGE_HANDLING (STATUS = ON | OFF) 来启用或禁用有害消息处理。

除此之外，Service Broker 还支持 Always On。

14.1.2　Service Broker 的作用

Service Broker 可帮助开发人员构建异步的松散耦合应用程序，在这些应用程序中彼此独立的组件相互配合来完成一项任务。这些应用程序组件会交换包含完成任务所需信息的消息。

1．会话

Service Broker 的核心就是发送和接收消息，是围绕发送和接收消息这一基本功能设计

的。在发送和接收消息前需要建立会话，每个消息都构成某个“会话”的一部分。会话是一个可靠的持久性通信信道。Service Broker 要求每个消息和会话都必须具有一个特定的类型，以帮助开发人员编写可靠的应用程序。

使用 Service Broker 相关的 T-SQL 语句就可以可靠地发送和接收消息。应用程序只需要向 Service Broker 提供的“服务”发送消息，这里的“服务”是一组相关任务的名称。应用程序是从“队列”中接收消息，这里的“队列”是一个内部表的视图。

在每个会话中，Service Broker 保证应用程序对每个消息只接收一次，并按照消息的发送顺序接收消息。这样可以防止消息被重复发送接收和乱序造成数据的错误。出于安全的考虑，Service Broker 使用基于证书的安全机制，有助于保护敏感消息并控制对服务的访问。

Service Broker 就像是邮政服务。用户通过发送信件（比如 E-mail）的方式与远方的朋友进行联络。邮政服务对电子邮件进行排序和投递。远方的朋友接收到的邮件也是按照发送的时间进行排序的，朋友阅读了邮件后撰写回信并发送新的信件，直至会话结束。这里电子邮件的传递是异步的，也就是说用户和他的朋友可以去做其他的事情，而不用在电脑旁等着对方回复邮件。在 Service Broker 中也是使用异步会话。Service Broker 消息的作用与信件相似。Service Broker 服务就好比是发件箱，而队列就像是收件箱，发送消息是对服务进行操作，接收消息是对队列进行操作。

当某应用程序向 Service Broker 服务发送消息时，它与会话另一端的应用程序的实现细节是隔离的。这样，就可以在不影响发送应用程序的情况下，对接收应用程序进行动态重新配置或用新代码替换它，甚至可以暂时关闭接收应用程序，唯一的影响是 Service Broker 将在接收应用程序重新启动之前，不断向其队列中添加新消息。

2. 消息排列和协调

Service Broker 中的队列与数据结构中的“队列”类似，遵从先进先出原则。在应用开发中比较常用的队列程序就是 MSMQ，但是在 Service Broker 中处理队列时，主要有如下两个方面与传统产品存在不同。

- Service Broker 队列集成在数据库中。
- 队列对相关消息进行协调和排序。

在 SQL Server 中集成队列有助于在日常的数据库维护和管理中将 Service Broker 包括在内。通常，管理员无须进行与 Service Broker 有关的日常维护任务。

Service Broker 保证程序按照消息的发送顺序而不是消息进入队列的顺序接收，并且会话中的每个消息只接收一次。Service Broker 框架提供了一个用于收发消息的简单 T-SQL 接口，以及用于确保消息传递和处理的一系列强有力保障。另外，Service Broker 保证两个队列读取器不能同时处理来自同一会话或同一相关会话组的消息。

在 Service Broker 会话中有两个端点。开始会话的一端称为发起方，另一端称为目标。每一端都有一个服务，即发起方服务和目标服务。在每个服务上都有一个关联的消息队列。典型的对话会话中消息的交换过程如下所示。

在发起方处：

（1）程序启动对话。

（2）程序生成一个消息，其中包含执行任务所需的数据。

（3）程序将此消息发送给目标服务。

在目标处：

（4）该消息被置于与目标服务关联的队列中。

（5）程序从队列接收消息并执行相应的任务。

（6）程序通过向发起方服务发送消息来进行响应。

在发起方处：

（7）该响应消息被置于与发起方服务关联的队列中。

（8）程序接收响应消息并对其进行处理。

这个循环不断重复，直至发起方由于没有更多要发送到目标的请求而结束会话。

SQL Server 2012 中的 Service Broker 支持为每个对话设置优先级，优先级从 10（高）～1（低）。可以将 Service Broker 系统配置为可提供不同级别的服务，从而确保优先级较低的任务不会妨碍优先级较高的任务。

编写消息传递应用程序时通常会遇到很多棘手的任务，这些困难任务包括消息协调、可靠的消息传递、锁定和启动队列读取器。Service Broker 可帮助处理这些任务，而不是由开发人员通过编程来进行处理，从而使数据库开发人员能够将精力集中在解决业务问题上。

3．事务性异步编程

Service Broker 不仅是异步的操作，同时还支持应用程序间的消息传递的事务性。Service Broker 消息传递的事务性要求如果某个事务回滚，则该事务中的所有 Service Broker 操作都将回滚，这包括发送和接收操作。在异步传递中，应用程序在数据库引擎处理传递的同时继续运行。

异步编程一般都是使用队列来实现。在很多异步编程的应用程序中都包含用做工作队列的表，其中包含当资源允许时所要完成的工作。队列可以为数据库应用程序带来以下两个优点。

- 应用程序可在将用户的工作请求置于队列中之后立即响应交互式用户。应用程序在响应之前不必等待完成所有与请求关联的工作。进入队列的请求在资源可用时会被处理，这使得数据库既可保持对交互式用户的响应，又可有效使用可用资源。
- 单个请求涉及的工作有时可以分为多个工作单元，每个单元都作为单独的事务进行处理。这时候，数据库应用程序可通过将请求置于队列中来启动每个工作单元。Service Broker 扩展了此概念，使应用程序可以将工作分散到不同计算机上的多个 Service Broker 实例中。

由开发人员编写应用程序来对队列中的项目进行正确排序和处理比较复杂，现在则可以使用数据库引擎中内置的 Service Broker 功能，简化成功实现数据库队列所需的编码工作。

4．支持松散耦合应用程序

Service Broker 支持松散耦合应用程序。松散耦合是指应用程序相互之间比较独立，这样的应用程序必须包含相同的交换消息定义，并且必须为服务之间的交互定义相同的总体结构，从而可以实现程序之间或者模块之间的通信。对于这种应用程序来说，它们都面对的是 Service Broker，应用程序不必知道会话中其他参与者的物理位置或实现细节。另外，此类应用程序不必同时运行，也不必在相同的 SQL Server 实例中运行，不必共享实

现细节。

14.1.3 Service Broker 的优点

使用 Service Broker 的功能来实现异步的数据库应用程序具有以下好处。

- 数据库集成提高了应用程序的性能，并简化了管理。
- 适用于简化的应用程序开发的消息排序和协调。
- 应用程序松散耦合提供了工作负荷灵活性。
- 相关消息锁定使一个应用程序的多个实例，可以对同一队列中的消息不必显式同步处理。
- 自动激活使应用程序可以随消息量进行调整。

1．数据库集成

前面提到的 Service Broker 的消息队列都是保存在数据库中的，其集成设计对应用程序性能和管理有所帮助。

采用传统的消息队列方式的异步编程若要进行事务性的消息传递，则必须编写程序处理事务成功或失败要对队列进行的操作，而与 SQL Server 集成可以进行事务性的消息传递，从而避免了外部分布式事务协调器的额外开销和复杂性。对 Service Broker 的操作通过 T-SQL 来实现，在一个数据库事务内，应用程序接收一个或多个消息，处理消息并发送答复消息。如果事务失败，所有工作都回滚，并且收到的消息会返回到队列中，以便可以尝试再次处理它。仅当应用程序提交事务之后，操作才会生效，而这些事务性的操作都是由 Service Broker 自动完成，无须人为干预。

使用 Service Broker 可以将数据、消息和应用程序逻辑都存放在数据库中，这样使管理变得更简单，因为应用程序的管理（灾难恢复、安全性、备份等）随之成为数据库日常管理的一部分，而且管理员也不必管理 3 或 4 个单独的组件。

在传统的消息传递系统中，消息存储区和数据库可以不一致。例如，当一个组件从备份中还原时，另一个组件也必须同时从备份中还原，否则消息存储区中的信息与数据库中的信息不匹配。Service Broker 将消息和数据保存在同一个数据库中，从而消除了不一致问题。

应用程序的消息传递和数据部分可以在 Service Broker 应用程序中使用相同的 SQL Server 语言和工具，这使开发人员在基于消息的编程中可以利用 SQL Server 编程实现消息队列。实现 Service Broker 服务的存储过程可以用 T-SQL 或一种 CLR 语言（比如 C#）来编写。而数据库外的程序使用 T-SQL 和类似的数据库编程接口，如 ADO.NET。

此外，数据库集成使自动资源管理成为可能。Service Broker 在 SQL Server 实例的上下文中运行，因此它监视准备从该实例中所有数据库传送的所有消息。这样使每个数据库在维护自己队列的同时，帮助 Service Broker 管理整个 SQL Server 实例中的资源使用情况。

2．排序和协调消息

在传统的消息传递系统中，应用程序要负责对消息进行排序，协调到达顺序可能不对的消息。例如，应用程序 A 发送消息 1、2 和 3。应用程序 B 接收并确认消息 1 和 3，但是

在接收消息 2 时出现错误。应用程序 A 重新发送消息 2，但是，现在该消息是在消息 1 和 3 后面收到的。使用传统的消息传递系统，开发人员或者必须编写应用程序，使消息顺序不出问题；或者在消息 2 到达之前暂时缓存消息 3，以便应用程序可以以正确的顺序处理这些消息。这两个解决方案既不明了也不简单。

使用传统的消息传递系统还存在另一个问题，即重复传递。在上面的示例中，如果应用程序 B 收到消息 2，但是返回给应用程序 A 的确认消息丢失，应用程序 A 将重新发送消息 2，这样将导致应用程序 B 收到消息 2 两次。应用程序需要编写相应代码识别出重复消息并放弃它，或者重新处理该重复消息，而不产生任何负面效果。同样，这两个方法也很难实现。

除了消息的排序和避免消息重复外，消息协调在传统上也是难以处理的问题。例如，某个应用程序可能向一个服务提交成百上千个请求。该服务并行处理这些请求，并且每处理完一个请求就立即返回一个响应。由于处理每个请求所花费的时间长短是不同的，因此应用程序收到响应的顺序与它发送传出消息的顺序不同。但是，为了正确处理响应，应用程序必须将每个响应与正确的传出消息相关联。在传统的消息传递系统中，每个应用程序都管理此关联，这使得应用程序的开发成本和复杂程度加大了。

Service Broker 通过自动处理消息顺序、唯一传递和会话标识解决上面提到的 3 个问题。在两个 Service Broker 端点间建立会话后，一个应用程序只对每个消息接收一次，并且以该消息的发送顺序接收它。不需要对应用程序额外编写代码，即可按照顺序处理这些消息，且仅处理一次。最后，Service Broker 自动将标识符包含在每个消息中。应用程序可以始终掌握特定消息属于哪个会话。

3．松耦合与工作负荷灵活性

通过 Service Broker 可以实现多个程序之间的松耦合。应用程序可以向队列发送一个消息，然后继续应用程序处理，而依靠 Service Broker 确保该消息到达目标。这种松耦合带来了计划上的灵活性。发起方可以发出多个消息，而多个目标服务可以并行处理这些消息。每个目标服务按照其自己的速度处理这些消息，具体情况取决于当前的工作负荷。

排队也可使系统更平均地分配处理任务，从而减少服务器所需的峰值容量。这样可以提高数据库应用程序的总体吞吐量和性能。例如，许多数据库应用程序在一天中的某些特定时段事务量激增，这使资源消耗加大并使响应时间延长。使用 Service Broker，可以将无须立即执行的业务放入队列中，系统在业务量减小，系统资源充足的时候再对队列中的业务进行处理。这些执行后台处理的应用程序可靠地处理这些事务一段时间，同时主入口应用程序继续接收新的业务事务。

在向 Service Broker 发送消息时，如果目标不是立即可用的，消息保留在发送数据库的传输队列中。Service Broker 会重试发送该消息直至成功发送为止，或者直至会话的生存期过期，这使得即使两个服务中的一个在会话期间的某个点不可用，会话也可以在两个服务之间可靠地继续。传输队列中的消息是数据库的一部分，即使实例发生故障转移或重新启动，Service Broker 也可以传递消息。

4．相关消息锁定

在传统的消息传递应用程序中，最难实现的任务之一是使多个程序并行读取同一队列。当多个程序或多个线程读取同一队列时，可能会以错误的顺序处理消息。假设有个传

统消息传递的订单处理应用程序。队列收到两个消息，消息 A 包含有关创建订单标题的指令，消息 B 包含有关创建订单行各项的指令。如果这两个消息分别由不同的应用程序实例移出队列，并且同时处理，则订单行项事务有可能尝试先提交，并因订单尚未存在而失败。这一失败进而导致事务回滚和消息重新入队，然后消息被再次处理，从而浪费了资源。对于这种情况，解决此问题的传统做法是：将消息 A 和消息 B 中的信息组合成一个消息。这种方法对两个消息来说简单明了，但是对于涉及需要协调几十个甚至几百个消息的系统来说，就无能为力了。

Service Broker 通过关联会话组中的相关会话来解决这个问题。对于同一个关联会话组中的消息，Service Broker 在处理其中的任一条消息时会自动锁定同一会话组中的其他所有消息，使这些消息只能由一个应用程序实例来接收和处理。同时，对于其他会话组中的消息则可以继续移出队列并处理。这样使得多个并行应用程序实例可以可靠而有效地工作，而不需要在应用程序中编写复杂的锁定代码。

5. 自动激活

Service Broker 最有用的功能之一是激活功能。激活功能使应用程序可以动态地调整自身，以匹配队列中到达的消息量。Service Broker 提供了一些功能，这些功能使在数据库中运行的程序和在数据库外运行的程序都可以利用激活。但是，Service Broker 不要求应用程序必须使用激活。

在 SQL Server 运行时，Service Broker 监视队列中的活动，以确定某个应用程序是否在接收所有含有可用消息的会话消息。当有工作需要由新的队列读取器来执行时，Service Broker 激活功能会启动一个新的队列读取器。Service Broker 监视队列中的活动来确定何时有工作需要由队列读取器来执行。当队列读取器的数量与传入的通信流量相匹配时，队列会定期进入一种状态，即队列或者为空，或者该队列中的所有消息都属于正由另一队列读取器处理的会话。如果队列在一段时间内没有达到此状态，则 Service Broker 会激活应用程序的另一个实例。

Service Broker 在激活程序上，对于在数据库中运行的程序和在数据库外运行的程序使用不同的激活方法。对于在数据库中运行的程序，Service Broker 启动队列所指定的存储过程的另一个副本。对于在数据库外运行的程序，Service Broker 生成一个激活事件。程序监视此事件以确定何时需要另一个队列读取器。

Service Broker 并不会主动停止其通过激活功能启动的程序。所以激活的应用程序都被编写为如果在特定的时间段内没有消息到达以供接收，则在这段时间过后程序将自动关闭。以这种方法设计的应用程序，使应用程序实例的数量可以随服务的通信流量更改而动态地增大或减小。此外，当系统重新启动时，应用程序会自动启动以读取队列中的消息。

自动激活是 Service Broker 的特色，传统的消息传递系统没有此行为，结果在某个给定的时间专用于特定队列的资源经常不是太多就是太少。

14.2　会话对象

在大多数消息系统中，消息（Message）是通信的基本单元。每条消息都是一个独立

的实体，应用程序逻辑的任务是跟踪发送和接收消息。另一方面，大多数业务事务由大量相关的步骤和操作组成。Service Broker 会话对象被设计用于管理需要很长时间才能完成的业务事务，使之更加可靠和简单。

14.2.1　消息类型

消息是在使用 Service Broker 的应用程序之间交换的信息。每个消息都是某个会话的一部分。消息有特定的类型，该类型由发送该消息的应用程序确定。每个消息都有唯一的会话标识，以及它在会话内的序列号。接收消息时，Service Broker 使用消息的会话标识和序列号来对其进行排序。

消息的内容由应用程序确定。Service Broker 在收到消息时会验证其内容以确保此内容对于该消息类型来说是有效的。在 SQL Server 中无论何种消息类型和消息的长度是多少，消息的内容都将以 varbinary(max)类型进行保存。所以，只要能够被转换为 varbinary(max)的数据都可以作为消息，应用程序通常根据约定和消息的类型来处理消息的内容。

基于 Service Broker 的应用程序之间通过互相发送组成会话的消息进行通信。在会话中的各参与者必须对每个消息的名称和内容取得一致。消息类型对象定义了消息类型的名称，并定义消息所包含的数据类型。消息类型保存在创建它的数据库中。在 SSMS 的对象资源管理器中，展开具体某个数据库节点下的“Service Broker”节点，其下第一个节点便是“消息类型”。

注意：在参与会话的每个数据库中应创建相同的消息类型。

每个消息类型在定义时可以指定 SQL Server 对该类型消息执行验证。可以让 SQL Server 验证消息是否包含有效的 XML、是否包含符合特定架构的 XML 或者消息中是否根本没有数据。对于任意数据或二进制数据，消息类型也可以指定 SQL Server 不验证消息的内容。

如果指定了要验证消息类型，当目标服务收到消息时，开始执行验证。如果消息的内容与指定的验证不匹配，则 Service Broker 会向发送该消息的服务返回一个错误消息。无论怎样指定验证，应用程序都必须在程序使用数据之前验证消息的内容是否适合它。

对于空消息类型，消息的正文一定不能包含数据。对于指定正确 XML 格式的消息类型，消息的正文必须为格式正确的 XML。对于指定符合特定架构集合的 XML 的消息类型，消息的正文必须包含格式正确的 XML，且该 XML 对集合中的架构之一有效。对于没有指定验证的消息类型，SQL Server 可接收任何消息内容，包括二进制数据、XML 或空消息。

如果未在 Service Broker 的 SEND 命令中指定消息类型，则 Service Broker 提供一个名为 DEFAULT 的内置消息类型。另外 Service Broker 中还包含用于报告错误和对话状态的系统消息类型。

14.2.2　管理消息类型

SQL Server 中使用 CREATE MESSAGE TYPE 命令创建消息类型，其语法如代码 14.1 所示。

代码 14.1　创建消息类型语法

```
CREATE MESSAGE TYPE message_type_name
    [ AUTHORIZATION owner_name ]
    [ VALIDATION = {  NONE
                   | EMPTY
                   | WELL FORMED XML
                   | VALID_XML WITH SCHEMA COLLECTION
                                             schema collection name
                  } ]
```

其中各参数的含义是：

- message_type_name 为要创建的消息类型的名称。在当前数据库中创建一条新消息，并归 AUTHORIZATION 子句定义的主体数据库所有。不能指定服务器、数据库和架构名称。message_type_name 最多可以有 128 个字符。
- AUTHORIZATION owner_name 将消息类型所有者设置为指定数据库用户或角色。如果当前用户为 dbo 或 sa，则 owner_name 可以是任何有效用户或角色的名称。否则，owner_name 必须是当前用户的名称，或者是当前用户对其拥有 IMPERSONATE 权限的用户的名称，或者是当前用户所属的角色的名称。如果省略此子句，则消息类型属于当前用户。
- VALIDATION 指定 Service Broker 对此类型消息正文的验证方式。如果未指定此子句，则验证默认为 NONE。
- NONE 指定不执行验证。消息正文可以包含数据，也可以为 NULL。
- EMPTY 指定消息正文必须为 NULL。
- WELL_FORMED_XML 指定消息正文必须包含格式正确的 XML。
- VALID_XML WITH SCHEMA COLLECTION schema_collection_name 指定消息正文必须包含符合指定架构集合中的某一架构的 XML。schema_collection_name 必须是现有 XML 架构集合的名称。

说明： 消息类型不能是临时对象。创建消息类型时允许使用以“#”开头的消息类型名称，但创建出来的仍然是永久对象。

例如要创建一个 XML 验证类型和一个不进行验证的消息类型，则对应的 SQL 脚本如代码 14.2 所示。

代码 14.2　创建消息类型

```
CREATE MESSAGE TYPE
      [//AWDB/1DBSample/RequestMessage]
      VALIDATION = WELL_FORMED_XML;     --XML 验证
CREATE MESSAGE TYPE
      [//AWDB/1DBSample/ReplyMessage]   --不验证
```

对于已经创建的消息类型，若要修改其验证方式可以使用 ALTER MESSAGE TYPE 命令，其语法格式如代码 14.3 所示。

代码 14.3　修改消息类型语法

```
ALTER MESSAGE TYPE message_type_name
   VALIDATION =
```

```
    { NONE
     | EMPTY
     | WELL FORMED XML
     | VALID_XML WITH SCHEMA COLLECTION schema_collection_name }
```

其中的 message_type_name 就是要修改的消息类型的名称。例如对于前面创建的消息类型，现在需要修改其验证方式为不进行验证，则对应的 SQL 脚本如代码 14.4 所示。

代码 14.4　修改消息类型

```
ALTER MESSAGE TYPE --修改消息类型
[//AWDB/1DBSample/RequestMessage]
VALIDATION = NONE; --修改为不验证
```

当一个消息类型不再使用时，可以使用 DROP MESSAGE TYPE 命令将其删除。例如删除前面创建的 2 个消息类型，则对应的 SQL 脚本如代码 14.5 所示。

代码 14.5　删除消息类型

```
DROP MESSAGE TYPE [//AWDB/1DBSample/RequestMessage]   --删除消息
DROP MESSAGE TYPE [//AWDB/1DBSample/ReplyMessage]
```

注意：消息类型在 Service Broker 中会被约定引用，消息类型被任何约定引用都将无法删除。

14.2.3　约定

约定是两个服务间关于每个服务该发送哪些消息才能完成特定任务的协议。约定用于定义应用程序完成特定任务时所用的消息类型，还用于确定会话的哪一端可以发送该类型的消息。约定的定义也保存在数据库中，并且与其对应的消息类型在同一个数据库中。

在参与会话的每个数据库中应创建相同的约定。例如，如果人力资源应用程序要验证雇员 ID，则请求验证的服务必须知道另一个服务需要的消息类型。发出请求的服务还必须知道自己需要接收哪些类型的消息，以便准备处理它们。

约定用于指定完成所需工作要用的消息类型。约定还指定会话中可使用每个消息类型的参与者。在约定中可以定义 3 种类型的端点来发送指定的消息类型：某些消息类型可由任一参与者发送，而其他消息类型则只限发起方或目标方进行发送。约定必须包含由发起方或任一参与者发送的消息类型。否则，发起方无法发起一个使用该约定的会话。

在 Service Broker 中还包含一个名为 DEFAULT 的内置约定。DEFAULT 约定只包含消息类型 SENT BY ANY。如果 BEGIN DIALOG 语句中没有指定约定，则 Service Broker 将使用 DEFAULT 约定。

14.2.4　管理约定

在 SQL Server 中使用 CREATE CONTRACT 命令用于创建约定。创建约定的语法如代码 14.6 所示。

代码 14.6　创建约定语法

```
CREATE CONTRACT contract_name
   [ AUTHORIZATION owner_name ]
      (  {   { message_type_name | [ DEFAULT ] }
          SENT BY { INITIATOR | TARGET | ANY }
       } [ ,...n] )
```

其中较常用的几个参数含义如下：

- contract_name 为要创建的约定的名称。
- message_type_name 为作为约定的一部分所包括的消息类型的名称。
- INITIATOR 指示只有会话的发起方才能发送指定消息类型的消息。启动会话的服务称为会话的“发起方”。
- TARGET 指示只有会话的目标才能发送指定消息类型的消息。接收由另一个服务启动的会话服务称为会话的目标。
- ANY 指示发起方和目标都可以发送此类型的消息。
- [DEFAULT]指示此约定支持默认消息类型的消息。默认情况下，所有数据库都包含名为 DEFAULT 的消息类型。此消息类型使用的验证为 NONE。

注意：*在创建约定时，如果使用默认的消息类型，则 DEFAULT 不是关键字，必须作为标识符进行分隔。*

例如对于前面创建的 2 个消息类型，其中一个消息类型只能被发起方使用，另外一个消息类型只能被目标使用，具体的 SQL 脚本如代码 14.7 所示。

代码 14.7　创建约定

```
CREATE CONTRACT [//AWDB/1DBSample/SampleContract]  --创建约定
      ([//AWDB/1DBSample/RequestMessage]           --指定发送方
       SENT BY INITIATOR,
       [//AWDB/1DBSample/ReplyMessage]             --指定目标
       SENT BY TARGET
      );
```

在创建了约定后将无法修改约定中的方向和消息类型。在约定中必须允许发起方发送消息，如果约定未包含至少一个 SENT BY ANY 或 SENT BY INITIATOR 消息类型，则 CREATE CONTRACT 语句将失败。

如果需要修改某个约定，只有将该约定删除后重新创建，删除约定使用 DROP CONTRACT 命令。例如要删除前面创建的约定，则对应的脚本为：

```
DROP CONTRACT [//AWDB/1DBSample/SampleContract]
```

如果有任何服务或会话优先级正在引用某个约定，则不能删除该约定。

14.2.5　队列

队列负责存储消息。Service Broker 收到某个服务的消息后，将该消息插入该服务的队列中。为了获取发送给该服务的消息，应用程序将接收来自队列的消息。Service Broker 管

理队列，并呈现一个类似于表的队列视图。

每个服务都可与一个队列相关联。当某个服务的消息到达后，Service Broker 将该消息放到与该服务关联的队列中。

在队列中每条消息就是其中的一行。行中可以包含消息的内容，以及消息类型、消息所针对的服务、所遵循的约定、所属的会话、对消息执行的验证、队列的内部信息等多种信息。应用程序使用消息行中的信息来唯一标识每条消息，并对其进行相应处理。

应用程序从该服务的队列接收消息。对于每个会话，队列中返回消息的顺序与发送方发送消息的顺序一致。由单个接收操作返回的所有消息均属于一个会话组会话的一部分。事实上，一个队列中包含若干组相关的消息，每个会话组对应其中一组消息。每次应用程序执行从队列接收消息的操作时，该队列都要返回一组相关的消息。应用程序可以选择是接收特定会话的消息还是特定会话组的消息。队列不按严格的先进先出顺序返回消息，而是以消息的发送顺序返回每个会话的消息。因此，应用程序无须包含恢复消息原始顺序的代码。

队列可以与存储过程相关联。在这种情况下，当队列中有要处理的消息时，SQL Server 激活存储过程。SQL Server 可以启动存储过程的多个实例，可启用实例数量的最大值是可以配置的。

14.2.6　管理队列

若要在数据库中创建一个新队列，则可以使用 CREATE QUEUE 命令。创建队列的语法如代码 14.8 所示。

代码 14.8　创建队列语法

```
CREATE QUEUE queue name
   [ WITH
     [ STATUS = { ON | OFF }  [ , ] ]
     [ RETENTION = { ON | OFF } [ , ] ]
     [ ACTIVATION (
         [ STATUS = { ON | OFF } , ]
           PROCEDURE NAME = stored procedure name ,
           MAX QUEUE READERS = max readers ,
           EXECUTE AS { SELF | 'user_name' | OWNER }
            ) ]
    ]
     [ ON { filegroup | [ DEFAULT ] } ]
```

其中几个常用的参数含义如下：

- queue_name 为要创建的队列的名称。此名称必须符合有关 SQL Server 标识符指南。
- STATUS（队列）指定队列是可用（ON）还是不可用（OFF）。如果队列不可用，则不能向队列添加或删除消息。可以以不可用状态创建队列，从而在使用 ALTER QUEUE 语句将该队列更改为可用之前，使消息无法到达该队列。如果省略此子句，则默认该队列可用。
- RETENTION 指定队列的保持期设置。如果 RETENTION=ON，则使用此队列的会话发送或接收的所有消息都将保存在队列中，直到会话结束为止。这样就可以保

留消息以便审核，或者用于在出现错误时执行补偿事务。如果未指定此子句，则默认的保留设置为 OFF。

- ACTIVATION 指定与为了处理此队列中的消息而必须启动的存储过程有关的信息。
- STATUS（激活）指定 Service Broker 是否启动存储过程。当 STATUS=ON 时，如果当前运行的过程数小于 MAX_QUEUE_READERS，并且消息抵达队列的速度比存储过程接收消息的速度快，则队列启动用 PROCEDURE_NAME 指定的存储过程。当 STATUS=OFF 时，队列不启动存储过程。如果未指定该子句，则默认为 ON。
- PROCEDURE_NAME=procedure_name 指定处理此队列中的消息时要启动的存储过程的名称。
- MAX_QUEUE_READERS=max_readers 指定队列同时启动的激活存储过程的最大实例数。max_readers 的值必须是 0～32 767 之间的数字。
- EXECUTE AS 指定用于运行激活存储过程的 SQL Server 数据库用户账户。SELF 指定存储过程以当前用户身份执行。'user_name'为执行存储过程时所用的用户名。OWNER 指定存储过程以队列的所有者身份执行。
- ON filegroup|[DEFAULT]指定从中创建此队列的 SQL Server 文件组。可使用 filegroup 参数标识文件组，也可使用 DEFAULT 标识符以使用 Service Broker 数据库的默认文件组。在此子句的上下文中，DEFAULT 不是关键字，必须作为标识符进行分隔。如果未指定文件组，该队列使用数据库的默认文件组。

例如要创建一个队列 Queue1，该队列是没有激活的，另外再创建一个队列 Queue2，消息进入队列时，队列将启动存储过程 sp1。该存储过程以用户 User1 的身份执行。该队列最多启动存储过程的 5 个实例，对应的 SQL 脚本如代码 14.9 所示。

代码 14.9　创建队列

```
CREATE QUEUE Queue1             --创建队列
WITH STATUS=OFF;                --队列状态为禁用
GO
CREATE QUEUE Queue2
WITH STATUS=ON,                 --队列状态为启用
ACTIVATION (
    PROCEDURE_NAME = sp1,       --激活存储过程 sp1
    MAX_QUEUE_READERS = 5,      --最多启用 5 个存储过程的实例
    EXECUTE AS 'User1' ) ;      --执行 sp1 存储过程的用户为 User1
```

队列创建后如果没有激活，可以通过 ALTER QUEUE 命令激活。修改队列的语法与创建队列的语法类似，如代码 14.10 所示。

代码 14.10　修改队列语法

```
ALTER QUEUE queue_name
   [ WITH
     [ STATUS = { ON | OFF }  [ , ] ]
     [ RETENTION = { ON | OFF } [ , ] ]
     [ ACTIVATION (
         [ STATUS = { ON | OFF } , ]
           PROCEDURE_NAME = stored_procedure_name ,
```

```
        MAX QUEUE READERS = max readers ,
        EXECUTE AS { SELF | 'user name' | OWNER }
         ) ]
   ]
```

其中 queue_name 就是要修改的队列名称，其他参数与 CREATE QUEUE 中的参数相同，这里不再赘述。例如要将前面创建的队列 Queue1 修改为激活，则对应的 SQL 脚本如代码 14.11 所示。

代码 14.11　修改队列

```
ALTER QUEUE Queue1
WITH STATUS=ON
```

修改队列时，当指定了激活存储过程的队列中包含消息时，如果将激活状态从 OFF 更改为 ON，则会立即激活该激活存储过程。如果将激活状态从 ON 更改为 OFF，则会阻止代理激活存储过程的实例，但不会停止正在运行的存储过程实例。

要删除队列，可以使用 DROP QUEUE 命令。如果有任何服务正在引用一个队列，则不能删除该队列。例如要删除创建好的队列 Queue2，则对应的脚本为：

```
DROP QUEUE Queue2
```

队列可以作为 SELECT 语句的目标，直接使用 SELECT 语句即可查询队列中的内容。但是，只能使用在 Service Broker 会话中运行的语句（如 SEND、RECEIVE 和 END CONVERSATION）来修改队列的内容。队列并不像表一样作为 INSERT、UPDATE、DELETE 或 TRUNCATE 语句的目标。例如要查询队列 Queue1 中的消息，则对应的查询语句为：

```
SELECT *
FROM Queue1
```

表 14.1 中列出了在查询队列时返回的队列中列的数据类型和说明。

表 14.1　队列中的列

列　　名	数 据 类 型	说　　明
Status	tinyint	消息的状态。0=已收到消息、1=就绪、2=尚未完成、3=已保留发送消息
Priority	tinyint	应用于消息的会话优先级
queuing_order	bigint	消息在队列中的序号
conversation_group_id	uniqueidentifier	此消息所属会话组的标识符
conversation_handle	uniqueidentifier	此消息所属会话的句柄
message_sequence_number	bigint	消息在会话中的序列号
service_name	nvarchar(512)	要进行会话的服务的名称
service_id	int	要进行会话的服务的 SQL Server 对象标识符
service_contract_name	nvarchar(256)	会话遵循的约定的名称
service_contract_id	int	会话遵循约定的 SQL Server 对象标识符
message_type_name	nvarchar(256)	对消息进行说明的消息类型的名称
message_type_id	int	对消息进行说明的消息类型的 SQL Server 对象标识符
Validation	nchar(2)	对消息所用的验证。E=空、N=无、X=XML
message_body	varbinary(MAX)	消息的内容

14.2.7　服务

Service Broker 中的服务是指某个特定业务任务或一组业务任务的名称。会话是发生在服务之间。发起会话后，Service Broker 使用服务名称将消息传递到数据库中的正确队列，另外还要执行会话的约定并确定新会话的远程安全性。

在服务定义中需要为每个服务指定一个队列用来存放传入消息。与服务相关的约定用于定义特定任务，该服务接收这些任务的新会话。因此，目标服务指定与服务进行会话必须遵循一个或多个约定。发起会话但不从其他服务接收新会话的服务不需要指定约定。如果服务可以接收使用 DEFAULT 约定的消息，则 DEFAULT 约定必须包含在服务定义中。

14.2.8　管理服务

在 SQL Server 中创建服务使用 CREATE SERVICE 命令，其语法如代码 14.12 所示。

代码 14.12　创建服务语法

```
CREATE SERVICE service_name
   [ AUTHORIZATION owner name ]
   ON QUEUE [ schema name. ]queue name
   [ ( contract_name | [DEFAULT] [ ,...n ] ) ]
```

其中的几个参数含义为：

- service_name 为要创建的服务的名称。
- AUTHORIZATION owner_name 将服务所有者设置为指定的数据库用户或角色。
- ONQUEUE [schema_name.]queue_name 指定接收服务消息的队列。该队列必须与服务在同一数据库中。如果未提供 schema_name，则架构为执行该语句用户的默认架构。
- contract_name 指定针对此服务的约定。服务程序使用指定的约定来启动与此服务的会话。如果未指定约定，则该服务可能仅启动会话。
- [DEFAULT]指定该服务可作为遵循 DEFAULT 约定的会话目标。这里的 DEFAULT 不是关键字，必须作为标识符进行分隔。DEFAULT 约定允许会话的双方发送 DEFAULT 类型的消息。DEFAULT 消息类型使用 NONE 验证。

例如要创建服务//AWDB/1DBSample/TargetService，该服务使用队列 Queue1，同时使用的是前面创建的约定//AWDB/1DBSample/SampleContract，则创建该服务的 SQL 脚本如代码 14.13 所示。

代码 14.13　创建服务

```
CREATE SERVICE                              --创建服务
   [//AWDB/1DBSample/TargetService]
   ON QUEUE Queue1                          --指定服务所在队列
   ([//AWDB/1DBSample/SampleContract]);--指定约定
```

服务公开与其关联的约定提供的功能，以便其他服务可使用该功能。CREATE

SERVICE 语句指定针对此服务的约定。一个服务只能是使用该服务指定的约定会话的目标。未指定约定的服务不会向其他服务公开任何功能。

注意：创建服务时一个服务可以指定多个约定，也可以不指定任何约定，不指定约定时该服务只能启动对话。

对于已经创建的服务，可以通过 ALTER SERVICE 命令为此服务指定新队列、添加或删除约定。ALTER SERVICE 的语法如代码 14.14 所示。

代码 14.14　修改服务语法

```
ALTER SERVICE service name
   [ ON QUEUE [ schema name . ]queue name ]
   [ ( < ADD CONTRACT contract name | DROP CONTRACT contract name >
   [ , ...n ] ) ]
```

其中的参数与创建服务时的参数相似，这里不再重复介绍。例如对于前面创建的服务，现在需要将该服务对应的队列修改为 Queue2，另外再去掉原来的约定，则对应的 SQL 脚本如代码 14.15 所示。

代码 14.15　修改服务

```
ALTER SERVICE [//AWDB/1DBSample/TargetService]          --修改服务
   ON QUEUE Queue2
   (DROP CONTRACT [//AWDB/1DBSample/SampleContract]);   --去掉指定约定
```

当 ALTER SERVICE 语句从某个服务中删除一条约定后，此服务便不可再作为使用该约定的会话目标。因此，Service Broker 将不允许使用该约定与此服务建立新会话。使用该约定的现有会话不受影响。

如果要删除服务则使用 DROP SERVICE 命令，例如要删除前面创建的服务的代码为：

```
DROP SERVICE [//AWDB/1DBSample/TargetService]
```

删除服务将从该服务使用的队列中删除该服务的所有消息。Service Broker 将向使用该服务的所有已打开会话的远程端发送错误。如果任何会话优先级引用了某个服务，则不能删除该服务。

14.3　会 话 对 话

在 Service Broker 中建立了各会话对象后，就可以启用会话发送和接收消息，在完成了整个会话对话后再结束会话。本节将主要介绍会话对话的过程和如何实现发起会话、发送消息、接收消息和结束会话。

14.3.1　对话过程

所有由 Service Broker 发送的消息都是会话的一部分。对话会话，或“对话”，是两个服务之间的会话。实际上，对话是两个服务之间可靠的、持久性双向消息流。

对话提供一次顺序（EOIO）消息传递功能。对话使用会话标识符和包含在每条消息中的序列号来标识相关的消息，并以正确的顺序传递这些消息。

对话会话有两个参与者。“发起方”发起会话，“目标”接收发起方发起的会话。参与者是否发起会话决定该参与者是否可以发送消息，如会话约定中所指定的。如图 14.1 所示为对话的消息流。

应用程序作为对话的一部分来交换消息。SQL Server 接收对话消息时，把消息放在对话队列中。应用程序接收来自队列的消息，并在必要时处理该消息。作为处理的一部分，应用程序可能会将消息发送到对话中的其他参与者。

对话中包含自动消息回执确认信息，以确保可靠的传递。Service Broker 将每条外发消息保存在传输队列中，直到消息被远程服务确认为止。有了这些自动确认信息，应用程序就不必再显式确认每条消息，从而可节省时间和资源。在可能的情况下，确认消息可包含在对话的返回消息中。

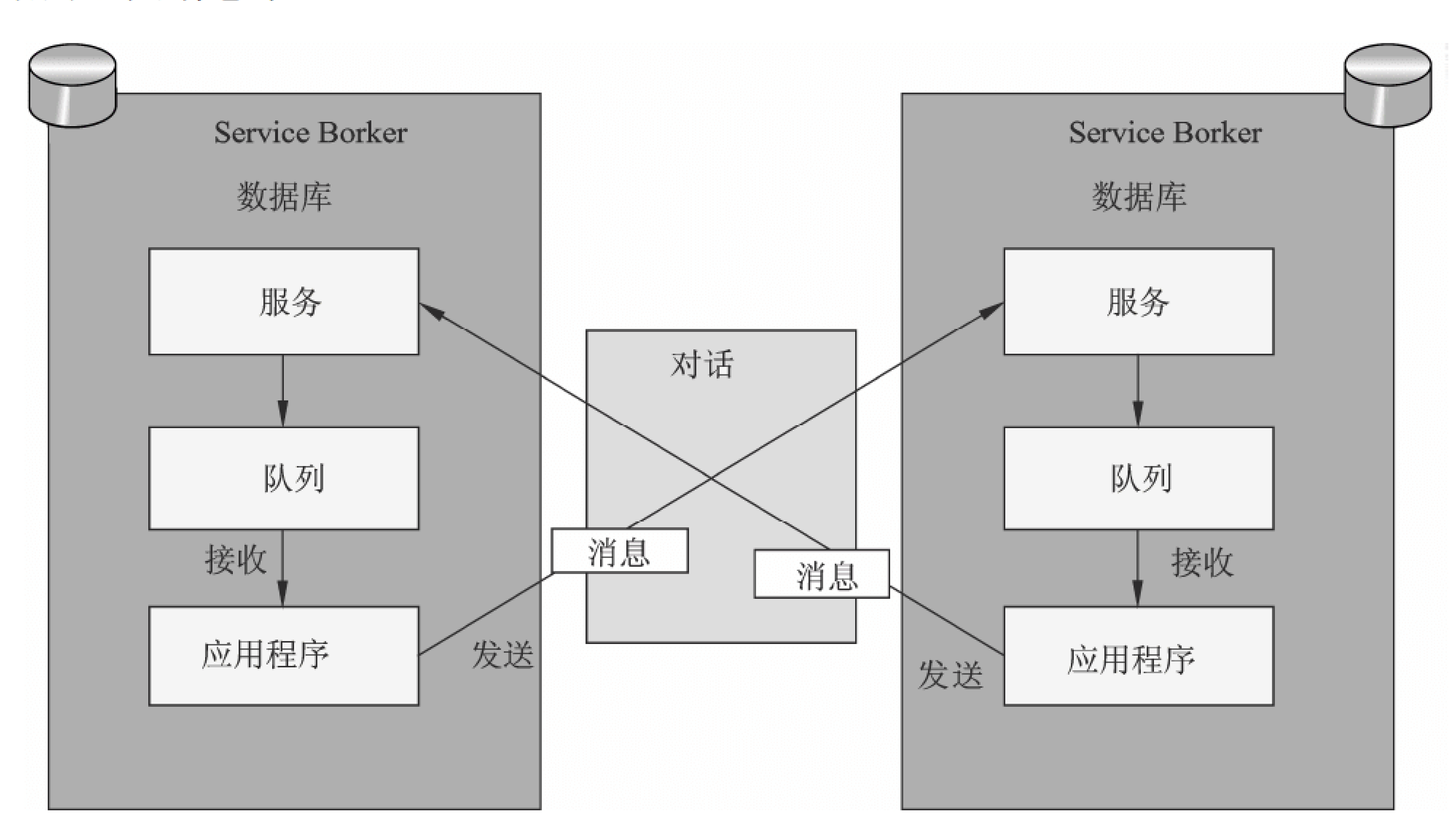


图 14.1　Service Broker 对话

注意：Service Broker 在内部处理确认消息。这些消息不出现在队列中，而且也不传递到应用程序中。

Service Broker 不将无法访问远程服务视为错误。远程服务无法访问时，Service Broker 保存该服务的消息，直到该服务可访问或对话生存期过期为止。

消息在对话生存期期间，可以在应用程序间进行交换。对话生存期从本地 SQL Server 实例创建对话时起，一直持续到应用程序显式结束该对话，或收到与该对话关联的错误消息时为止。每个参与者均有责任在应用程序收到指示错误的消息，或结束会话的消息时显式结束会话。在大多数服务中，其中一个参与者负责通过无错误地结束会话，指示会话完成且成功。此操作是由目标执行还是由发起方执行取决于会话的目的。

起始应用程序发起对话时，它的本地 Service Broker 为该对话创建一个会话端点。目标应用程序的本地 Service Broker 在其实例收到该对话中的第一个消息时，为该对话创建一个会话端点。

对话还可以保证会话生存期不超出指定的时限。起始应用程序可以选择指定对话的最长生存期。本地 Service Broker 和远程 Service Broker 都会跟踪此生存期。如果对话在到达最大生存期时仍处于活动状态，会话的每一端都会将一个超时错误消息放在服务队列中，并拒绝新的对话消息。会话的生存时间从不会超出在对话开始时所建立的最大生存期。请注意，虽然会话结束之后应用程序仍可接收该会话的消息，但不会有该会话的新消息到达，应用程序也无法再发送有关该会话的消息。

应用程序负责通过显式结束对话来指示它们已处理完该对话。Service Broker 从不自动结束对话。对话将保留在数据库中，直到应用程序显式结束会话为止。因此，即使在对话超时或 Broker 报告错误时，会话中的各参与者也必须显式发出 ENDCONVERSATION 语句。

利用会话计时器，应用程序可以在特定时间接收消息。会话计时器到期时，SQL Server 在启动会话计时器的端点将一个会话消息插入到会话队列中。应用程序可以将会话计时器用于任何目的。会话计时器的一个常见用途是响应远程服务响应的延迟时间。另一个常见用途是创建以设定间隔向远程服务发送消息的服务。例如，服务可以利用会话计时器，每隔几分钟报告一次 SQL Server 的当前状态。应用程序还可以使用会话计时器在特定时间激活存储过程，这使 Service Broker 可以支持预定活动。

会话中的每个参与者在每个会话中都可设置一个会话计时器。会话计时器不与其他参与者共享，而且它也不影响会话的生存期。在计时器到期时，本地 Service Broker 将一个超时消息添加到本地服务队列中。超时消息的类型名称为 http://schemas.Microsoft.com/SQL/ServiceBroker/DialogTimer。

14.3.2　发起和结束会话

前面已经介绍了会话相关的对象和进行会话对话的过程，现在就可以使用 T-SQL 语句发起和结束会话。发起会话使用 BEGIN DIALOG 命令，其语法如代码 14.16 所示。

代码 14.16　发起会话语法

```
BEGIN DIALOG [ CONVERSATION ] @dialog_handle
   FROM SERVICE initiator_service_name
   TO SERVICE 'target service name'
       [ , { 'service broker guid' | 'CURRENT DATABASE' } ]
   [ ON CONTRACT contract name ]
   [ WITH
   [  { RELATED CONVERSATION = related conversation handle
      | RELATED CONVERSATION GROUP = related conversation group id } ]
   [ [ , ] LIFETIME = dialog lifetime ]
   [ [ , ] ENCRYPTION = { ON | OFF }  ] ]
```

其中重要的几个参数含义是：

- @dialog_handle 是一个变量，用于为 BEGIN DIALOG CONVERSATION 语句返回的新对话存储系统生成的对话句柄。该变量的类型必须为 uniqueidentifier。
- initiator_service_name 指定启动对话的服务。指定的名称必须是当前数据库中服务的名称。为发起方服务指定的队列将接收由目标服务返回的消息，以及 Service Broker 为此会话创建的消息。

- target_service_name 指定启动对话时的目标服务。target_service_name 的类型为 nvarchar(256)。Service Broker 会逐字节进行比较，以便与 target_service_name 字符串匹配。换言之，比较时将区分大小写，且不考虑当前的排序规则。
- contract_name 指定此会话遵循的约定。当前数据库中必须有该约定。如果目标服务不接收遵循指定约定的新会话，则 Service Broker 将返回针对该会话的错误消息。如果省略此子句，则会话将遵循名为 DEFAULT 的约定。
- LIFETIME=dialog_lifetime 指定对话将保持打开状态的最长时间。为使对话成功完成，两个端点都必须在生存期内显式结束对话。dialog_lifetime 的值必须以秒表示。生存期的类型为 int。如果未指定 LIFETIME 子句，则对话的生存期为 int 数据类型的最大值。
- ENCRYPTION 指定将此对话发送和接收的消息向 SQL 实例外发送时，是否必须对消息加密。必须加密的对话是安全对话。如果 ENCRYPTION=ON，但未配置支持加密所需的证书，则 Service Broker 将返回针对该会话的错误消息。ENCRYPTION=OFF 时，如果为 target_service_name 配置了远程服务绑定，则使用加密；否则，发送消息时不加密。如果未使用此子句，则默认值为 ON。

说明： *在 BEGIN DIALOG 语句中 FROM SERVICE 指定发起方是使用的 SQL Server 名称，而在 TO SERVICE 中指定目标却使用的是字符串，这是为了准备在 Service Broker 中可以和 SQL Server 服务器之外的服务进行通信。因此，目标参数是一个字符串。*

例如，现在有创建好的服务“//AWDB/1DBSample/InitiatorService”和“//AWDB/1DBSample/TargetService”，这 2 个服务之间使用约定“//AWDB/1DBSample/SampleContract”进行不加密的通信，则对应的发起会话的 SQL 脚本如代码 14.17 所示。

代码 14.17　发起会话示例

```
DECLARE @InitDlgHandle UNIQUEIDENTIFIER;                 --定义句柄
BEGIN DIALOG @InitDlgHandle                              --开启会话
FROM SERVICE [//AWDB/1DBSample/InitiatorService] --指定从哪个服务到哪个服务
TO SERVICE '//AWDB/1DBSample/TargetService'
ON CONTRACT [//AWDB/1DBSample/SampleContract]            --会话中使用的约定
WITH ENCRYPTION = OFF;                                   --不加密会话
```

会话在发起后并最终完成时结束，结束会话使用 END CONVERSATION 命令。结束会话的语法如代码 14.18 所示。

代码 14.18　结束会话语法

```
END CONVERSATION @conversation_handle
   [   [ WITH ERROR = failure_code DESCRIPTION = 'failure_text' ]
     | [ WITH CLEANUP ]
```

其中@conversation_handle 是要结束的会话的会话句柄。failure_code 指定失败代码，类型为 int。失败代码是用户定义代码，包含在发送给会话另一方的错误消息中。失败代码必须大于 0。failure_text 指定了错误消息。failure_text 的类型为 nvarchar(3000)。失败文本是用户定义文本，包含在发送给会话另一方的错误消息中。

在开发 Service Broker 应用程序时，会出现大量的孤立对话，即使在应用程序结束后，这些对话仍然存在。WITH CLEARUP 选项可以删除无法正常完成的会话一方的所有消息和目录视图条目，而不会向会话的另一方通知此清除操作。清除会话时，SQL Server 将删除会话端点、传输队列中该会话的所有消息及服务队列中该会话的所有消息。管理员可以使用此选项删除无法正常完成的会话。

注意：不要在 Service Broker 应用程序的代码中使用 WITH CLEANUP。如果在接收端确认收到消息之前运行 END CONVERSATION WITH CLEANUP，则发送端点将再次发送该消息。这样可能导致再次运行该对话。

例如对应前面创建的会话，若要结束该会话则执行：

```
END CONVERSATION @InitDlgHandle
```

14.3.3　发送和接收消息

14.3.2 节中介绍了如何发起和结束会话，但是在发起会话后并没有执行任何的操作，本节将主要介绍如何发送和接收会话中的消息。

在会话中发送消息使用 SEND 命令，而接收消息使用 RECEIVE 命令。SEND 命令负责向会话的队列中添加一条消息，而 RECEIVE 命令则负责从队列中移除一条消息。SEND 命令的语法如代码 14.19 所示。

代码 14.19　SEND 语法

```
SEND ON CONVERSATION @conversation_handle
   [ MESSAGE TYPE message_type_name ]
   [ ( message_body_expression ) ]
```

其中@conversation_handle 指定消息所属的会话的句柄。@conversation_handle 必须包含一个有效的会话标识符。

message_type_name 指定已发送的消息的类型，必须将此消息类型包含在此会话使用的服务约定中。约定必须允许从此会话方发送此消息类型。例如，会话目标只能发送在约定中指定为 SENT BY TARGET 或 SENT BY ANY 的消息。如果省略此子句，则消息类型为 DEFAULT。

message_body_expression 提供一个表示消息主体的表达式。message_body_expression 是可选的。但如果存在 message_body_expression，则表达式必须是可以转换为 varbinary(max) 的类型，且不能为 NULL。如果省略该子句，则消息主体为空。

如果目标队列可用，而且是位于 SEND 命令所在的数据库实例中，则消息会直接插入目标队列中。否则，消息会被插入到本地数据库的传送队列中。在使用 SEND 命令发送消息时，如果指定的会话不存在或者已经关闭，则发送消息将会失败。例如发起一个会话，在该会话中发送一条 XML 消息，则对应的 SQL 脚本如代码 14.20 所示。

代码 14.20　发起会话并发送消息

```
DECLARE @InitDlgHandle UNIQUEIDENTIFIER,@Msg xml;
BEGIN DIALOG @InitDlgHandle--开启会话
FROM SERVICE [//AWDB/1DBSample/InitiatorService]
```

```
TO SERVICE '//AWDB/1DBSample/TargetService'
ON CONTRACT [//AWDB/1DBSample/SampleContract]
WITH ENCRYPTION = OFF;
--以上是发起会话，接下来就是在该会话中发送消息
SET @Msg='<Say>Hello World</Say>';      --定义发送消息的内容
SEND ON CONVERSATION @InitDlgHandle     --在会话中发送消息
    MESSAGE TYPE [//AWDB/1DBSample/RequestMessage](@Msg); --指定消息类型
```

现在已经可以查询 Queue1 队列，发现发送的消息已经在队列中。队列的查询与表的查询相同，直接使用 SELECT 命令即可。而在 SSMS 中查询队列则更简单，直接在对象资源管理器中右击队列，选择“选择前 1000 行”选项即可。

队列中有 14 列，但是 SEND 命令却只有 3 个参数，其他列都是根据会话的句柄从 Service Broker 的元数据中获取的。从本质上来说，SEND 命令就是队列表的 INSERT 命令。现在消息已经成功发送到队列中，接下来就是使用 RECEIVE 命令接收消息。RECEIVE 命令的语法如代码 14.21 所示。

代码 14.21　RECEIVE 语法

```
[ WAITFOR ( ]
   RECEIVE [ TOP ( n ) ]
      * | { column_name | [ ] expression } [ [ AS ] column_alias ] |
column_alias = expression [ ,...n ]
      FROM queue_name
      [ INTO table_variable ]
      [ WHERE {  conversation_handle = @conversation_handle
             | conversation_group_id = @conversation_group_id } ]
[ ] ] [ , TIMEOUT timeout ]
```

RECEIVE 命令的语法相对于 SEND 命令要复杂得多，其中各个参数的含义是：

- WAITFOR 指定如果当前没有消息，则 RECEIVE 语句将等待消息到达队列。
- TOP(n)指定要返回的最大消息数。如果未指定该子句，则返回所有满足语句条件的消息。
- *,column_name,expression,column_alias 指定结果集中的列。
- FROM queue_name 指定包含要检索的消息的队列。
- INTO table_variable 指定 RECEIVE 将消息放入到表变量。表变量具有的列数必须与消息中的列数相同。表变量中每列的数据类型必须可隐式转换为消息中对应列的数据类型。如果未指定 INTO，则消息将作为结果集返回。
- WHERE 指定用于已收到消息的会话或会话组。如果省略，则从下一个可用会话组返回消息。
- conversation_handle=@conversation_handle 指定用于已收到消息的会话。提供的@conversationhandle 必须为 uniqueidentifer 类型，或者为可转换为 uniqueidentifier 的类型。
- conversation_group_id=@conversation_group_id 指定用于已收到消息的会话组。提供的@conversation_group_id 必须为 uniqueidentifier 类型，或者为可转换为 uniqueidentifier 的类型。
- TIMEOUT timeout 指定语句等待消息的时间（以毫秒为单位）。该子句只能与 WAITFOR 子句一起使用。如果未指定该子句，或者超时值为–1，则等待时间将为

无限长。如果超时时间已到，RECEIVE 将返回一个空结果集。

例如要接收 Queue1 中的一条消息，并将该消息的内容打印出来，则对应的 SQL 语句如代码 14.22 所示。

代码 14.22 接收消息并打印

```
DECLARE @Msg nvarchar(100);
RECEIVE TOP(1)
@Msg=convert(nvarchar(100),message body)
FROM Queue1
PRINT @Msg
```

这条 RECEIVE 语句的问题是，如果队列中没有任何可用的消息，则该语句将立刻返回，没有对任何数据进行处理。如果需要一有消息时就执行 RECEIVE 操作，则需要指定 WAITFOR 语句。使用了 WAITFOR 语句后，如果队列为空，则 RECEIVE 语句将进行等待，直到超时或者接收到消息。

例如创建一个无限长时间等待接收消息的脚本，该脚本将接收到的内容放入一个表变量中，则对应的 SQL 脚本如代码 14.23 所示。

代码 14.23 接收消息放入表变量

```
DECLARE @tbMsg table(msg nvarchar(100),receiveTime datetime)
WAITFOR(                    --不超时的等待接收消息
RECEIVE TOP(1)              --接收一条消息
convert(nvarchar(100),message_body),getdate()
FROM Queue1                 --消息队列
INTO @tbMsg                 --将消息放入变量
)
SELECT * FROM @tbMsg        --查询接收到的消息
```

注意： SEND 和 RECEIVE 关键字必须是开始的第一个命令，因为这两个命令并不是 ANSI SQL 的一部分，并不会被分析器认为是终结符。为了确保分析器知道 SEND 和 RECEIVE 是开始的一个新命令，需要在 SEND 和 RECEIVE 之前的命令添加分号（;）结束。但是在 WAITFOR 语句中的 RECEIVE 语句前却不添加分号。

14.3.4 会话组

在编写异步的消息应用程序时，最困难的处理就是多个应用程序或者多个线程从一个队列中同时读取消息。例如一个队列中存在 1 号和 2 号消息，多线程程序同时读取了这 2 个消息进行处理，有可能 2 号消息比 1 号消息先处理完成，这样就会破坏消息的顺序。一种简单的解决办法就是，在应用程序中对于每个队列只允许接收一个线程。这种方法的伸缩性不好，但是可以维护数据的一致性。

在 Service Broker 中采用了会话组的概念来维护数据的一致性，也保证了程序的伸缩性。会话组标识一组相关的会话。应用程序通过会话组可以轻松地协调特定业务任务所涉及的会话。每个会话都属于一个会话组。每个会话组都与一个特定的服务相关联，并且，组中的所有会话都是面向该服务或者来自该服务的。一个会话组可以包含任意数量的会话。

使用会话组后，可以保证对与特定业务任务相关联的消息提供一次顺序访问。当应用程序发送或接收消息时，SQL Server 锁定当前消息所属的会话组，保持该锁定直到事务完

成。因此，即使在多线程应用程序中也能保证一次只有一个会话可以接收该会话组的消息。

由于会话组可以包含多个会话，因此应用程序可以使用会话组标识与同一业务任务相关的消息，并同时处理这些消息。

默认情况下，会话与会话组之间是一对一的关系。会话组不在会话的各参与者之间共享。因此，会话中的每个参与者可以根据需要自由地对会话进行分组。应用程序无须服务为其提供任何特殊支持，即可管理服务间的复杂交互。

在 BEGIN DIALOG 命令中有 3 种方法可以把会话分到会话组，可以指定已经属于某个会话组的会话句柄，也可以指定一个会话组 ID，还可以使用自己创建的 GUID 来创建新的会话组。例如创建一个会话 A，然后再创建一个会话 B，将会话 B 添加到会话 A 所在的会话组中，则对应的 SQL 语句如代码 14.24 所示。

代码 14.24　将一个会话添加到会话组

```
DECLARE @A UNIQUEIDENTIFIER,@B UNIQUEIDENTIFIER;
BEGIN DIALOG @A     --开启一个会话
FROM SERVICE [//AWDB/1DBSample/InitiatorService]
TO SERVICE '//AWDB/1DBSample/TargetService'
ON CONTRACT [//AWDB/1DBSample/SampleContract];
--接下来再开启一个会话
BEGIN DIALOG @B
FROM SERVICE [//AWDB/1DBSample/InitiatorService]
TO SERVICE '//AWDB/1DBSample/TargetService'
ON CONTRACT [//AWDB/1DBSample/SampleContract]
WITH  RELATED_CONVERSATION=@A   --说明与第一个会话是同一个会话组
```

这种情况下创建的会话，只知道 A 和 B 会话是在同一个会话组中，而不知道具体的会话组 ID，因此可以在创建会话时指定一个会话组 ID，使用该会话组 ID 创建一个会话组，其他会话则根据该会话 ID 添加到该会话组中。

例如定义一个 GUID 变量@GroupID，在发起会话 A 时将通过该 ID 创建一个会话组，而发起会话 B 的语法与会话 A 相同，但是由于指定 ID 的会话组已经存在，所以是将 B 会话添加到指定 ID 的会话组中。具体 SQL 脚本如代码 14.25 所示。

代码 14.25　指定会话组 ID 创建会话组和添加会话

```
DECLARE @A UNIQUEIDENTIFIER,@B UNIQUEIDENTIFIER,@GroupID UNIQUEIDENTIFIER;
SET @GroupID=NEWID();   --定义会话组的标识
BEGIN DIALOG @A         --开启一个会话
FROM SERVICE [//AWDB/1DBSample/InitiatorService]
TO SERVICE '//AWDB/1DBSample/TargetService'
ON CONTRACT [//AWDB/1DBSample/SampleContract]
WITH RELATED_CONVERSATION_GROUP=@GroupID; --创建一个会话组
--接下来再开启一个会话
BEGIN DIALOG @B
FROM SERVICE [//AWDB/1DBSample/InitiatorService]
TO SERVICE '//AWDB/1DBSample/TargetService'
ON CONTRACT [//AWDB/1DBSample/SampleContract]
WITH RELATED_CONVERSATION_GROUP=@GroupID;  --将会话添加到会话组中
```

在接收消息时，为了方便，SQL Server 允许应用程序在没有收到消息的情况下锁定下一个可用的会话组。使用 GET CONVERSATION GROUP 语句，应用程序可以锁定一个会话组，并在处理消息前还原状态，其语法如代码 14.26 所示。

代码 14.26　GET CONVERSATION GROUP 语法

```
[ WAITFOR ( ]
   GET CONVERSATION GROUP @conversation_group_id
      FROM <queue>
[ ] ] [ , TIMEOUT timeout ]
```

例如要从队列 Queue1 中锁定下一个会话组，获得会话组的 ID，则对应的 SQL 语句如代码 14.27 所示。

代码 14.27　锁定 Queue1 队列中的下一会话组

```
DECLARE @conversation_group_id UNIQUEIDENTIFIER ;
WAITFOR (
 GET CONVERSATION GROUP @conversation_group_id --锁定队列中的下一会话组
    FROM Queue1
) ;
```

14.3.5　单个数据库的会话

在 Service Broker 的应用中，最简单的情况就是在单个数据库中进行会话对话。例如要创建一个简单的 Hello World 的 Service Broker 应用，向该 Service Broker 中传入用户姓名，Service Broker 在接收到用户姓名后返回用户姓名+“，Hello World”的字符串。具体操作过程如下。

（1）创建用于该 Service Broker 应用的数据库 BrokerTest1，使用创建数据库的 SQL 语句如下：

```
CREATE DATABASE BrokerTest1
GO
```

（2）修改 BrokerTest1 数据库的属性，启用该数据库的 Service Broker，具体 SQL 脚本如代码 14.28 所示。

代码 14.28　启用 Service Broker

```
USE master;
GO
ALTER DATABASE BrokerTest1
      SET ENABLE_BROKER;--启用 Service Broker
```

（3）创建发送消息和回复消息的消息类型，使用默认的不进行消息正文验证的方式，具体创建 SQL 脚本如代码 14.29 所示。

代码 14.29　创建消息类型

```
USE BrokerTest1;
GO
CREATE MESSAGE TYPE[//BrokerTest1/Sample1/RequestMessage]--创建发生消息类型
CREATE MESSAGE TYPE [//BrokerTest1/Sample1/ReplyMessage] --创建接收消息类型
```

（4）创建约定 Contract1，在该约定中指定发送消息类型和接收消息类型，如代码 14.30 所示。

代码 14.30　创建约定

```
CREATE CONTRACT [//BrokerTest1/Sample1/Contract1]
--创建约定指定发送类型和接收类型
(
    [//BrokerTest1/Sample1/RequestMessage] SENT BY INITIATOR,
    [//BrokerTest1/Sample1/ReplyMessage] SENT BY TARGET
);
```

（5）创建目标队列和服务，具体脚本如代码 14.31 所示。

代码 14.31　创建目标队列和服务

```
CREATE QUEUE TargetQueue;  --创建目标队列
GO
CREATE SERVICE [//BrokerTest1/Sample1/TargetService]  --创建服务
ON QUEUE TargetQueue([//BrokerTest1/Sample1/Contract1]);
GO;
```

（6）创建发起方的队列和服务，具体脚本如代码 14.32 所示。

代码 14.32　创建发起方队列和服务

```
CREATE QUEUE InitiatorQueue;   --创建发起方队列
GO
CREATE SERVICE [//BrokerTest1/Sample1/InitiatorService]   --创建发起方服务
ON QUEUE InitiatorQueue;
```

（7）发起会话，向目标发送消息，消息内容为一个用户姓名，具体脚本如代码 14.33 所示。

代码 14.33　发起会话并发送消息

```
DECLARE @InitDlgHandle UNIQUEIDENTIFIER;
BEGIN DIALOG @InitDlgHandle            --开启会话
    FROM SERVICE [//BrokerTest1/Sample1/InitiatorService]
    TO SERVICE N'[//BrokerTest1/Sample1/TargetService]'
    ON CONTRACT [//BrokerTest1/Sample1/Contract1];--这个分号是必需的
SEND ON CONVERSATION @InitDlgHandle    --在会话中发送消息
    MESSAGE TYPE [//BrokerTest1/Sample1/RequestMessage](N'ZengYi');
                                                      --发生消息内容
```

（8）接收目标队列中的消息内容，然后对该内容进行处理，最后再将处理的结果发送给发起方，具体脚本如代码 14.34 所示。

代码 14.34　接收消息并发送答复

```
DECLARE @RecvReqDlgHandle UNIQUEIDENTIFIER;
DECLARE @RecvReqMsg NVARCHAR(100);
DECLARE @RecvReqMsgName sysname;
WAITFOR             --等待接收消息
( RECEIVE TOP(1)    --接收一条消息
   @RecvReqDlgHandle = conversation_handle,   --将接收的消息赋给变量
   @RecvReqMsg = message_body,
   @RecvReqMsgName = message type name
  FROM TargetQueue);
```

```
IF @RecvReqMsgName = N'//BrokerTest1/Sample1/RequestMessage' --判断消息正
                                                               确性
BEGIN
    DECLARE @ReplyMsg NVARCHAR(100);
    SET @ReplyMsg = @RecvReqMsg+N',Hello World';               --回复消息
    SEND ON CONVERSATION @RecvReqDlgHandle
        MESSAGE TYPE [//BrokerTest1/Sample1/ReplyMessage](@ReplyMsg);
    END CONVERSATION @RecvReqDlgHandle;
END
```

（9）处理答复并结束会话。在发起方收到答复后将答复内容打印出来，然后结束整个会话，具体脚本如代码 14.35 所示。

代码 14.35　处理答复并结束会话

```
DECLARE @RecvReplyMsg NVARCHAR(100);
DECLARE @RecvReplyDlgHandle UNIQUEIDENTIFIER;
WAITFOR             --等待回复
( RECEIVE TOP(1)    --读取一条回复
   @RecvReplyDlgHandle = conversation_handle,
   @RecvReplyMsg = message_body
  FROM InitiatorQueue
);
END CONVERSATION @RecvReplyDlgHandle;  --结束会话
PRINT @RecvReplyMsg--输出回复的内容
```

14.4　Service Broker 网络会话

Service Broker 除了在单个数据库中进行异步队列处理外，还可以在 SQL Server 实例与 SQL Server 实例之间进行会话。在 Service Broker 进行跨实例的会话中，需要使用到 Service Broker 端点、协议、路由等功能，本节将主要介绍 Service Broker 如何进行网络会话。

14.4.1　Service Broker 端点

SQL Server 使用 Service Broker 端点（Endpoint）使 Service Broker 与 SQL Server 实例外部进行通信。

在前面介绍数据库镜像的配置时使用到了端点，端点是一个 SQL Server 对象，代表 SQL Server 进行网络通信。每个端点支持一种特定的通信类型。例如，HTTP 端点使 SQL Server 可以处理特定的 SOAP 请求。Service Broker 端点将 SQL Server 配置为通过网络发送和接收 Service Broker 消息。

在默认情况下，SQL Server 的实例并没有包含任何 Service Broker 端点。因此，默认情况下 Service Broker 是不能通过网络发送或接收消息的。要进行 Service Broker 网络会话，则必须创建 Service Broker 端点。只有创建 Service Broker 端点后，才能向 SQL Server 实例外部发送消息或从该实例外部接收消息。一个实例可以只包含一个 Service Broker 端点。

Service Broker 若要进行网络会话，则会话的安全性至关重要。Service Broker 有一套完备的安全机制，使不同 SQL Server 实例承载的服务之间可以安全地通信。Service Broker

安全机制依赖于“证书”。在 Service Broker 中创建端点时使用证书建立远程数据库的凭据，另外还需要将操作从远程数据库映射到本地用户。

在端点中使用证书和映射的本地用户后，Service Broker 将实例外部的操作映射到 SQL Server 数据库中的这个用户上。在创建端点时指定的证书必须满足下列要求才能用于 Service Broker 安全性。

- 密钥模块必须小于 2048。
- 证书总长度必须小于 32 KB。
- 必须指定主题名称。
- 必须指定有效日期。
- 密钥长度必须为 64 位的倍数。

创建端点之前先是创建证书，创建证书使用 CREATE CERTIFICATE 命令。例如现在要在服务器 A 和服务器 B 之间进行 Service Broker 网络会话，则先在 A 服务器上创建证书，具体脚本如代码 14.36 所示。

代码 14.36　创建证书

```
USE master ;
GO
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'your password'--创建主密钥
GO
CREATE CERTIFICATE A cert                                         --创建证书
WITH SUBJECT = 'A certificate for database Service Broker',       --证书说明
START DATE = '01/01/2008',                                      --证书开始日期
EXPIRY_DATE = '10/31/2099' ;                                    --证书失效日期
GO
```

创建证书后接下来便可创建 Service Broker 端点。创建 Service Broker 端点使用 CREATE ENDPOINT 命令，其语法如代码 14.37 所示。

代码 14.37　CREATE ENDPOINT 语法

```
CREATE ENDPOINT endPointName [ AUTHORIZATION login ]
[ STATE = { STARTED | STOPPED | DISABLED } ]
AS { HTTP | TCP } (
   <protocol_specific_arguments>
        )
FOR { SOAP | TSQL | SERVICE_BROKER | DATABASE_MIRRORING } (
   <language_specific_arguments>
        )
```

这里只是简化的语法，实际上使用 CREATE ENDPOINT 命令可以创建多种类型的端点，不同的端点有不同的配置，对应的语法也不一样。其中比较重要的几个参数的意义是：

- endPointName 是所创建端点的已分配名称。在更新或删除端点时使用。
- AUTHORIZATION login 指定被分配了新建端点对象的所有权的有效 SQL Server 或 Windows 登录账户。如果没有指定 AUTHORIZATION，默认情况下调用方将成为新建对象的所有者。
- STATE={STARTED|STOPPED|DISABLED}指定端点创建时的状态。如果在创建端点时未指定状态，则默认值为 STOPPED。

- ❑ STARTED 表示端点已启动并在积极地侦听连接。
- ❑ DISABLED 表示端点被禁用。在该状态下，服务器不侦听端点端口，也不对使用端点的任何尝试请求进行响应。
- ❑ STOPPED 表示端点被停止。在该状态下，服务器侦听端口请求但向客户端返回错误。
- ❑ AS{HTTP|TCP}指定要使用的传输协议。
- ❑ FOR{SOAP|TSQL|SERVICE_BROKER|DATABASE_MIRRORING}指定负载类型。

例如要在 A 服务器上创建一个端点，该端点使用 TCP 协议 4022 端口，使用的是前面创建的证书 A_cert 的认证方式，则对应的创建端点的脚本如代码 14.38 所示。

代码 14.38　创建 Service Broker 端点

```
CREATE ENDPOINT Endpoint_Broker_A
    STATE = STARTED
    AS TCP (
       LISTENER_PORT=4022                          --通信中所使用的端口
       , LISTENER_IP = ALL                         --允许所有 IP
    )
FOR SERVICE_BROKER(
       AUTHENTICATION = CERTIFICATE A_cert  --使用前面创建的证书
       , ENCRYPTION = REQUIRED ALGORITHM RC4
);
```

说明：根据约定，Service Broker 通常使用端口 4022 进行 Broker 间通信。但是这并不是必需的，最终使用的端口是在创建端点时指定的。

14.4.2　路由

Service Broker 进行网络会话时使用路由来确定向何处传递消息。当服务在会话中发送消息时，SQL Server 使用路由来确定将接收该消息的服务。当该服务响应时，SQL Server 再次使用路由来确定发出消息的服务。Service Broker 中的路由由如下 3 个基本组成成分组成。

- ❑ Service name：该路由指定为其寻址的服务的名称。此名称必须完全匹配 BEGINDIALOG 命令中的 Service Name。
- ❑ Broker instance identifier：为要将消息发送到特定数据库的唯一标识符。这是此路由指向数据库的 sys.databases 表行中的 service_broker_guid 列。
- ❑ Network address：它是实际的计算机地址，或者是将路由限制到本地计算机的关键字，或是指示传输层从服务名称推导出地址的关键字。网络地址可以是服务宿主 Broker 的地址，也可以是转发 Broker 的地址。

在一个数据库中可以定义多个路由。在确定会话路由时，SQL Server 将 BEGIN DIALOG CONVERSATION 语句中指定的服务名称和 Broker 实例标识符与路由中指定的服务名称和 Broker 实例标识符进行匹配，以寻找合适的路由。

在匹配过程中遵从没有提供服务名称的路由与所有服务名称匹配，没有提供 Broker 实例标识符的路由与所有 Broker 实例标识符匹配。当多个路由匹配会话时，SQL Server 选

择其一。

在查找到匹配的路由并选择了其中一个后，接下来就是定位目标服务。当选定的路由指定“LOCAL”作为网络地址时，Service Broker 就在自己的实例中查找服务。如果实例中不存在该服务，则 Service Broker 可能会返回上一步选择其他路由。

创建路由使用 CREATE ROUNT 命令，其语法如代码 14.39 所示。

代码 14.39　CREATE ROUNT 语法

```
CREATE ROUTE route name [ AUTHORIZATION owner name ]
WITH
   [ SERVICE NAME = 'service name', ]
   [ BROKER INSTANCE = 'broker instance identifier' , ]
   [ LIFETIME = route lifetime , ]
   ADDRESS =  'next hop address'
   [ , MIRROR_ADDRESS = 'next_hop_mirror_address' ]
```

其中几个重要的参数意义如下：

- route_name 是要创建的路由的名称。该名称中不能指定服务器、数据库和架构名称。route_name 必须是有效的 sysname。
- SERVICE_NAME='service_name'指定此路由指向的远程服务的名称。service_name 必须与远程服务使用的名称完全匹配。Service Broker 将进行逐字节比较来匹配 service_name。也就是说，这种比较区分大小写，并且不考虑当前的排序规则。如果省略 SERVICE_NAME，则此路由将与任何服务名称匹配，但比指定 SERVICE_NAME 的路由的匹配优先级低。服务名称为'SQL/Service Broker/Broker Configuration'的路由是指向 Broker Configuration Notice 服务的。指向此服务的路由不能指定 Broker 实例。
- BROKER_INSTANCE='broker_instance_identifier'指定承载目标服务的数据库。broker_instance_identifier 参数必须是远程数据库的 Broker 实例标识符。如果省略了 BROKER_INSTANCE 子句，则此路由将与任何 Broker 实例匹配。如果会话没有指定 Broker 实例，则匹配任何 Broker 实例的路由的匹配优先级，高于具有显式 Broker 实例的路由的匹配优先级。对于指定了 Broker 实例的会话，具有 Broker 实例的路由的优先级高于匹配任何 Broker 实例的路由的优先级。
- LIFETIME=route_lifetime 以秒为单位指定 SQL Server 在路由表中保留路由的时间。在生存期结束后，相应的路由即过期，SQL Server 在为新会话选择路由时将不再考虑该路由。如果省略此子句，则 route_lifetime 为 NULL，并且路由始终不过期。
- ADDRESS='next_hop_address'指定此路由的网络地址。next_hop_address 按以下格式指定 TCP/IP 地址：TCP://{dns_name|netbios_name|ip_address}:port_number。

例如在 BrokerTest1 数据库中创建一个路由，该路由指定了服务“//AWDB/1DBSample/TargetService”的路由网络地址是“TCP://MyTargetComputer:4022”，则对应的 SQL 脚本如代码 14.40 所示。

代码 14.40　创建路由

```
USE BrokerTest1;
CREATE ROUTE InstTargetRoute          --创建路由
```

```
WITH SERVICE NAME =
N'//AWDB/1DBSample/TargetService',          --路由对应的服务
ADDRESS = N'TCP://MyTargetComputer:4022';   --使用的协议、端口等
```

注意： 如果是创建本地路由，只将 ADDRESS 设置为“LOCAL”即可，不需要指定协议和端口。

无论用户是否创建了路由，每个数据库都包含一个路由 AutoCreatedLocal，该路由匹配任何的服务名称和 Broker 实例，并指定消息应在当前实例中传递。对于会话的发起方和目标在同一 SQL Server 实例中的简单情况而言，无须其他路由。正是因为该路由的重要性，所以一般是为每个服务分别创建一个路由，这样有助于保护 Auto Created Local 路由，使其不会被修改或删除。

14.5　小　　结

Service Broker 是在 Microsoft SQL Server 2005 中才开始使用的新技术，它可帮助数据库开发人员生成安全、可靠且可伸缩的应用程序。本章主要讲解了 Service Broker 的原理和如何创建 Service Broker 的会话对话，在 Service Broker 中使用消息实现应用程序之间信息的交换。最后还介绍了使用 Service Broker 进行数据库实例之间的网络会话。

第 15 章　空间数据类型

空间数据是表示有关几何对象的物理位置和形状的信息。SQL Server 支持两种空间数据类型，分别是 geometry 数据类型和 geography 数据类型，这两种数据类型在 SQL Server 中都是作为.NET 公共语言运行时（CLR）数据类型实现的。本章将主要介绍空间数据类型的基础知识及使用。

15.1　空间数据类型简介

从 SQL Server 2008 开始就增加了一个亮点就是对空间数据类型的支持，使得 SQL Server 从纯粹的关系数据库系统发展成“关系型数据库＋空间数据引擎”。SQL Server 继续延续对空间数据类型的支持。本节将对空间数据类型的基础做讲解。

15.1.1　空间数据类型概述

在 SQL Server 2012 中，地理空间数据类型分为两种，即欧式（平面）几何 geometry 和地理空间（椭圆体）几何 geography。欧式几何以坐标 XY 表示，geometry 数据类型适用于 SQL 规范的开放地理空间联盟（OGC）简单特征 1.1.0 版。而地理空间就是使用经度和纬度来表示。

geometry 和 geography 数据类型支持 11 种空间数据对象或实例类型。但是，这些实例类型中只有 7 种“可实例化”，可以在数据库中创建并使用这些实例（或可对其进行实例化）。这些实例的某些属性由其父级数据类型派生而来，使其在 GeometryCollection 中区分为 Points、LineStrings、Polygons 或多个 geometry 或 geography 实例。

在空间数据对象中有 3 个基本元素：点、线、面。对应的在空间数据类型中也是这 3 种类型 Point、LineString、Polygon，以及各自的集合 MultiPoint、MultiLineString、MultiPolygon，另外还有一个它们的混合集合 GeometryCollection，混合集合中可以包含多个 Point、LineString 和 Polygon。很容易看出 MultiPoint、MultiLineString、MultiPolygon 是 GeometryCollection 的一种特殊情况。这几种类型就是 geometry 和 geography 数据类型的 7 种可实例化类型。空间数据对象的层次结构如图 15.1 所示。

两种空间数据类型的行为经常很相似，但在数据存储方式和操作方式上存在某些重要的差别。在平面（或平面球）系统中，均以相同的量度单位为坐标测量距离和面积。如果使用 geometry 数据类型，（2，2）和（5，6）之间的距离为 5 个单位，与所用的单位无关。

在椭圆体（或圆球）系统中，坐标以经度和纬度的度数给定。但是，即使测量可能依据的是 geography 实例的空间引用标识符（SRID），长度和面积的测量单位也通常为米或

平方米。geography 数据类型最常见的量度单位为米。

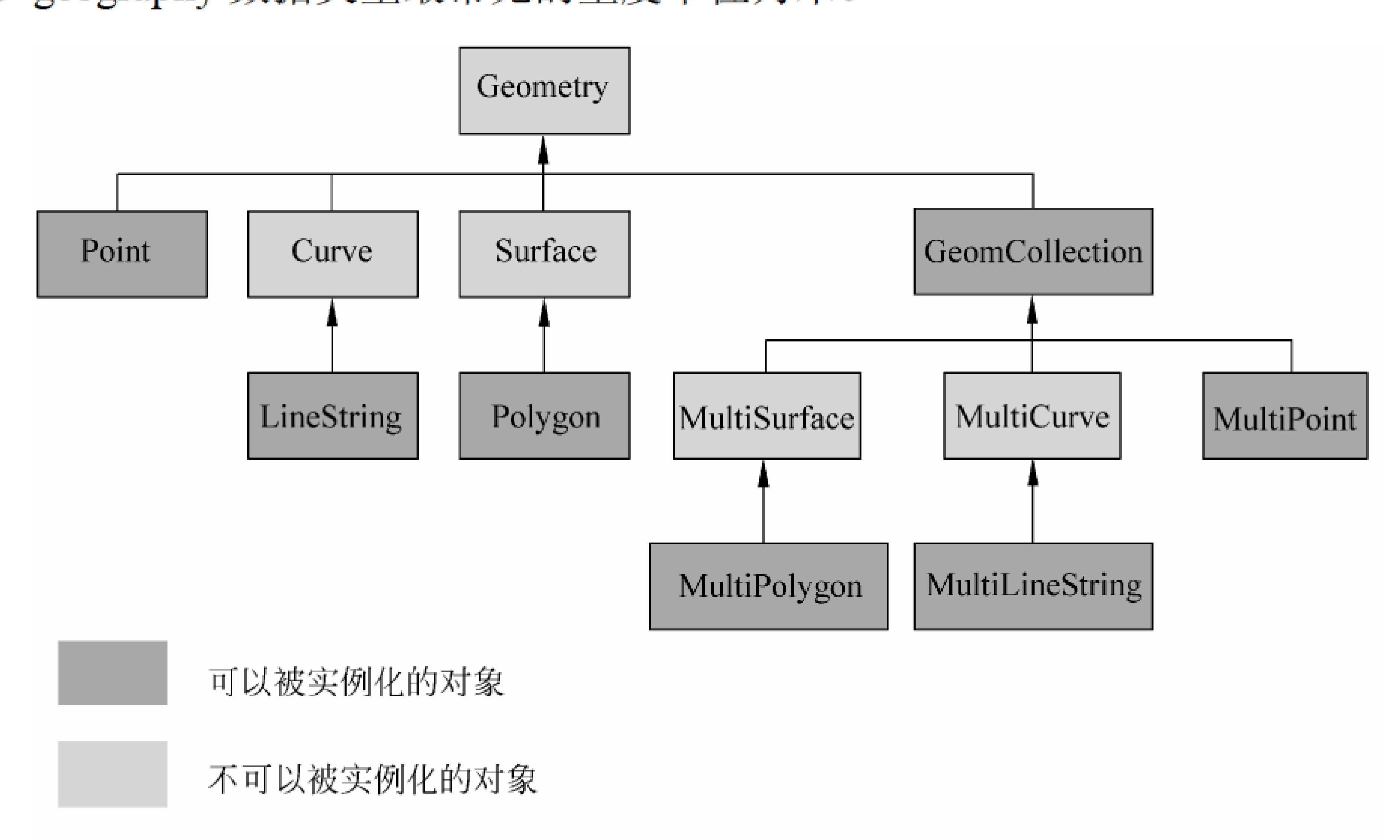


图 15.1　空间数据对象层次结构

在平面系统中，多边形的环方向并非重要因素。例如，((0，0)，(10，0)，(0，20)，(0，0)) 描述的多边形与 ((0，0)，(0，20)，(10，0)，(0，0)) 描述的多边形相同。SQL 规范的 OGC 简单特征未规定环顺序，并且 SQL Server 不会强制环的顺序。

在椭圆体系统中，多边形无意义，或者模糊不清，没有方向。例如，赤道周围的环是否描述了北半球或南半球？如果使用 geography 数据类型存储空间实例，必须指定环的方向并准确地描述实例的位置。SQL Server 2012 在使用 geography 数据类型时具有以下限制。

- 每个 geography 实例必须能够容纳在单个半球的内部。任何大于半球的对象都无法存储。
- 使用开放地理空间联盟（OGC）熟知文本（Well-KnownText，WKT）或熟知二进制（Well-KnownBinary，WKB）表示形式，并且会产生大于一个半球的对象的任何 geography 实例都会引发一个 ArgumentException 异常。
- 如果方法的结果不能容纳在单个半球内部，则需要输入两个 geography 实例的 geography 数据类型方法（如 STIntersection()、STUnion()、STDifference() 和 STSymDifference()），将返回 Null。如果输出超过单个半球，STBuffer()也将返回 Null。

在地理数据类型中，外环和内环并不重要。在 SQL 规范的 OGC 简单特征中讨论了外环和内环，但此差别对 SQL Server geography 数据类型来说几乎毫无意义：多边形的任何环都可以作为外环。

15.1.2　WKT 简介

WKT 是 Well-Known Text 的简称，中文翻译为熟知文本。WKT 是一个为在地图上描述矢量几何对象的文本标记语言。几何对象中点、线、面和体都可以用 WKT 来表示。不

同的几何尺寸，可以储存在一个几何集合。

WKT 几何用于整个 OGC 规范，并在目前的 SQL Server 2012 中得到了很好的支持。例如代码 15.1 所示的 WKT 示例都可以转换为 SQL Server 2012 中的空间对象。

代码 15.1　一些 WKT 示例

```
POINT(610)
LINESTRING(34,1050,2025)
POLYGON((11,51,55,15,11),(22,32,33,23,22))
MULTIPOINT(3.55.6,4.810.5)
MULTILINESTRING((34,1050,2025),(-5-8,-10-8,-15-4))
MULTIPOLYGON(((11,51,55,15,11),(22,32,33,23,22)),((33,62,64,33)))
GEOMETRYCOLLECTION(POINT(46),LINESTRING(46,710))
```

除了使用 WKT 表示空间对象外，WKB（Well-Known Binary）也可以用于在 SQL Server 2008 中表示空间对象。WKB 主要是用来传输和存储空间对象信息到数据库。

除了 WKT 和 WKB 两个格式外，SQL Server 2012 还支持通过 GML（几何对象的 XML 表示形式）来生成空间对象。

15.1.3　空间引用标识符

每个空间实例都有一个空间引用标识符（SRID）。SRID 对应基于特定椭圆体的空间引用系统，可用于平面球体或圆球映射。空间列可包含具有不同 SRID 的对象。然而，在使用 SQL Server 空间数据方法对数据执行操作时，仅可使用具有相同 SRID 的空间实例。从两个空间数据实例派生的任何空间方法的结果，仅在这两个实例具有相同的 SRID（该 SRID 基于相同的用于确定实例坐标的量度单位、数据和投影）时才有效。SRID 最常见的量度单位为米或平方米。

如果两个空间实例的 SRID 不相同，则对这两个实例使用 geometry 或 geography 数据类型方法后，结果将返回 NULL。例如，若要以下谓词返回非 NULL 结果，两个 geometry 实例（geometry1 和 geometry2）必须具有相同的 SRID：

```
geometry1.STIntersects7c18f5be-5a29-422e-8ca7-d8a5f38e03f5(geometry2)=1
```

说明：空间引用标识系统是由 European Petroleum Survey Group（EPSG）standard（欧洲石油测绘组（EPSG）标准）定义的，它是为绘图、测绘及大地测量数据存储而开发的一组标准。该标准归石油天然气生产商（OGP）测绘和定位委员会所有。

SQL Server 中 geometry 实例的默认 SRID 为 0。利用 geometry 空间数据，执行计算不需要空间实例指定 SRID，因此，实例可驻留在未定义的平面空间。若要在 geometry 数据类型方法的计算中指明未定义的平面空间，SQL Server 数据库引擎需要使用 SRID0。

SQL Server 支持基于 EPSG 标准的 SRID。必须使用 geography 实例支持 SQL Server 的 SRID 执行计算，或将方法用于地域空间数据。SRID 必须与 sys.spatial_reference_systems 目录视图中显示的 SRID 中的一个匹配。如前所述，在使用 geography 数据类型对空间数据执行计算时，结果将取决于在创建数据时使用的是哪个椭圆体，因为为每个椭圆体都分配

了一个特定空间引用标识符（SRID）。

对 geography 实例使用方法时，SQL Server 使用等于 4326 的默认 SRID，它将映射到 WGS84 空间引用系统。如果要使用 WGS84（或 SRID4326）之外的某个空间引用系统中的数据，需要确定地域空间数据的特定 SRID。

15.1.4　空间类

由于空间数据类型基于 CLR 集成实现，所以了解空间数据对象对应的类及其之间的关系，有助于我们更好地理解和使用空间数据类型。如图 15.2 所示为 OGC 标准中空间对象之间的关系图。

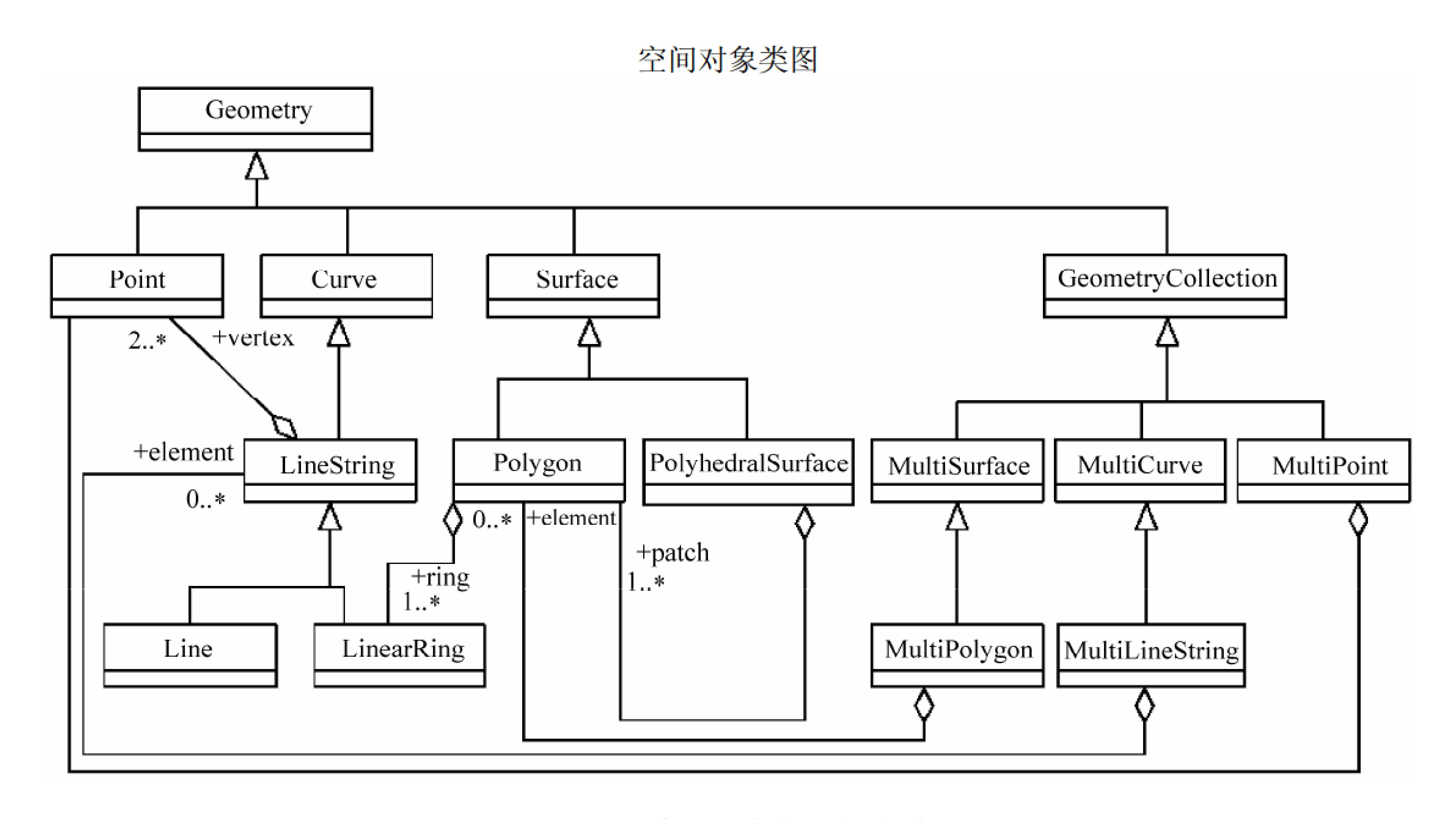


图 15.2　空间对象之间的关系

从图 15.2 中可以看出所有空间对象都继承于 Geometry 对象。在 Geometry 对象中定义的属性和方法可以在所有空间对象中使用。MultiPoint、MultiLineString 和 MultiPolygon 都继承至 GeometryCollection 对象，所以这 3 个对象都可以转换为 GeometryCollection 对象，也可以认为这 3 个对象是 GeometryCollection 对象的一种特例。

同时从图 15.2 中还可以看到空间对象之间的关系：一个 LineString 对象至少包含两个 Point 对象；一个 MultiLineString 对象包含 0 到多个 LineString 对象等。

15.2　geometry 几何数据类型

几何数据类型 geometry 又叫平面空间数据类型，用于表示欧几里得（平面）坐标系中的数据。geometry 是作为 SQL Server 中的.NET 公共语言运行时（CLR）数据类型实现的。本节主要讲解 geometry 数据类型的使用。

15.2.1　Point 点的使用

在 SQL Server 空间数据中，Point 是表示单个位置的零维对象，可能包含 Z（仰角）和 M（量度）值。SQL Server 提供了通过 WKT、WKB 和 GML 来实例化 Point 对象的静态方法。所谓静态方法就是指直接通过类，而不需要实例化对象来调用的方法。创建 Point 对象的方法包括：

- ❑ STGeomFromText ('geography_tagged_text'，SRID)；
- ❑ STPointFromText ('point_tagged_text'，SRID)；
- ❑ Parse ('geography_tagged_text')；
- ❑ STGeomFromWKB ('WKB_geography'，SRID)；
- ❑ STPointFromWKB ('WKB_point'，SRID)。

其中 STGeomFromText()、STPointFromText()和 Parse()方法通过 WKT 构造 Point 对象，另外的 STGeomFromWKB()和 STPointFromWKB()方法通过 WKB 构造 Point 对象。

SRID 是一个 int 类型，它表示希望返回的 geometry Point 实例的空间引用 ID（SRID）。

实际上 STGeomFromText()、Parse()和 STGeomFromWKB()方法是在 Geometry 对象中声明的，也就是说除了 Point 对象外，所有 Geometry 对象都可以通过这几个方法来构造。

WKT 中使用 Point 关键字表示点，其格式为：

```
Point(X,Y[,M,Z])
```

其中 X 表示点的 X 坐标值，Y 表示点的 Y 坐标值，M 表示仰角，Z 表示量度。M 和 Z 不是必需的。

例如要构造一个点（1，2），使用 WKT 构造该 Point 对象的脚本如代码 15.2 所示。

代码 15.2　通过 WKT 构造 Point 对象

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POINT (1 2)', 0);   --定义点对象
SET @g = geometry::STPointFromText('POINT (1 2)',0);
SET @g = geometry::Parse('PoINT (1 2)');
```

注意：WKT 是不区分大小写的，写成 POINT、Point、PoINT 等格式都可以。

构造了 Point 对象后便可使用该对象实例的属性和方法。通过 Geometry 实例可以返回实例的 WKT 和 WKB，以下都是在 Geometry 类中定义，可以在任何 Geometry 类型实例中调用的方法。

- ❑ STAsText()；
- ❑ ToString()；
- ❑ STAsBinary()。

其中 STAsText()和 ToString()方法返回实例的 WKT，STAsBinary()方法返回实例的 WKB。例如要返回一个 Point 对象的 WKT 和 WKB，对应的脚本如代码 15.3 所示。

代码 15.3　返回实例的 WKT 和 WKB

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POINT (1 2)', 0);
```

```
SELECT @g.ToString()    --返回 WKT
SELECT @g.STAsBinary()  --返回对象的 WKB
```

返回的结果为：

```
POINT (1 2)
0x0101000000000000000000F03F0000000000000040
```

针对 Point 对象，系统提供了 STX、STY、Z 和 M 这 4 个属性，分别表示 Point 对象的 X 坐标、Y 坐标、仰角和量度。例如构造一个点，其 X 坐标为 1，Y 坐标为 2，仰角为 3，量度为 4，获得其属性如代码 15.4 所示。

代码 15.4　返回 Point 实例的 X、Y、Z 和 M

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POINT (1 2 3 4)', 0);
SELECT @g.STX;  --查询对象下的各个属性
SELECT @g.STY;
SELECT @g.Z;
SELECT @g.M;
```

注意：如果在构造 Point 对象时未指定 Z 和 M，则返回的对应属性为 NULL。在构造时 Z 和 M 值也可以显式指定为 NULL。

15.2.2　MultiPoint 点集的使用

MultiPoint 是零个或更多个点的集合。构造 MultiPoint 对象除了使用通用的 STGeomFromText、Parse 和 STGeomFromWKB 方法外，还可以使用以下方法：

- STMPointFromText ('multipoint_tagged_text'，SRID)；
- STMPointFromWKB ('WKB_multipoint'，SRID)。

MultiPoint 的 WKT 格式为：

```
MultiPoint ( (X1 Y1) [,(X2 Y2)]...)
```

其中（X1 Y1）表示其中的一个点，每个点之间使用逗号隔开。点的声明除了 XY 之外，还可以声明 Z 和 M，写成（X Y Z M）的格式。例如要构造一个 MultiPoint 对象，该对象中包含了 3 个点，则对应的脚本如代码 15.5 所示。

代码 15.5　构造 MultiPoint 对象

```
DECLARE @g geometry;
SET @g = geometry::STMPointFromText('MultiPoint((0 0),(1 1 2 3),(3 4))',
0);
```

在构造 MultiPoint 对象时可以忽略点的括号，直接使用逗号分隔每个点，如代码 15.6 所示。

代码 15.6　构造 MultiPoint 对象

```
DECLARE @g geometry;
SET @g = geometry::STMPointFromText('MultiPoint(0 0,1 1 2 3,3 4)', 0);
```

构造好的 MultiPoint 对象可以调用其实例方法 STAsText()和 ToString()返回该对象的 WKT。除此之外，若要获得实例中点的个数，可以使用 STNumGeometries()方法。

例如构造一个由 3 个点组成的 MultiPoint 对象，调用 STNumGeometries()方法获得其中点的数目如代码 15.7 所示。

代码 15.7　使用 STNumGeometries()方法

```
DECLARE @g geometry;
SET @g = geometry::STMPointFromText('MultiPoint(0 0,1 1 2 3,3 4)', 0);
SELECT @g.STNumGeometries()
```

系统将返回结果为 3。

注意：STNumGeometries()方法是在 Geometry 对象中定义的，所以对于每个 Geometry 对象都可以使用，对于 Point 对象，其返回的结果是 1。若实例为空，则该方法返回 0。

15.2.3　LineString 线的使用

LineString 是一个一维对象，表示一系列点和连接这些点的线段。一个 LineString 实例必须由至少两个非重复点组成，也可以为空。在 LineString 对象中有如下几个概念需要了解。

- 简单的：线中没有交叉的部分被称为简单的；否则称为不简单的。
- 闭合的：线的首尾相连被称为闭合的；否则称为非闭合的。

如图 15.3 所示，其中图 1 是简单非闭合的线，图 2 为不简单非闭合的线，图 3 为简单闭合的线，图 4 为不简单闭合的线。

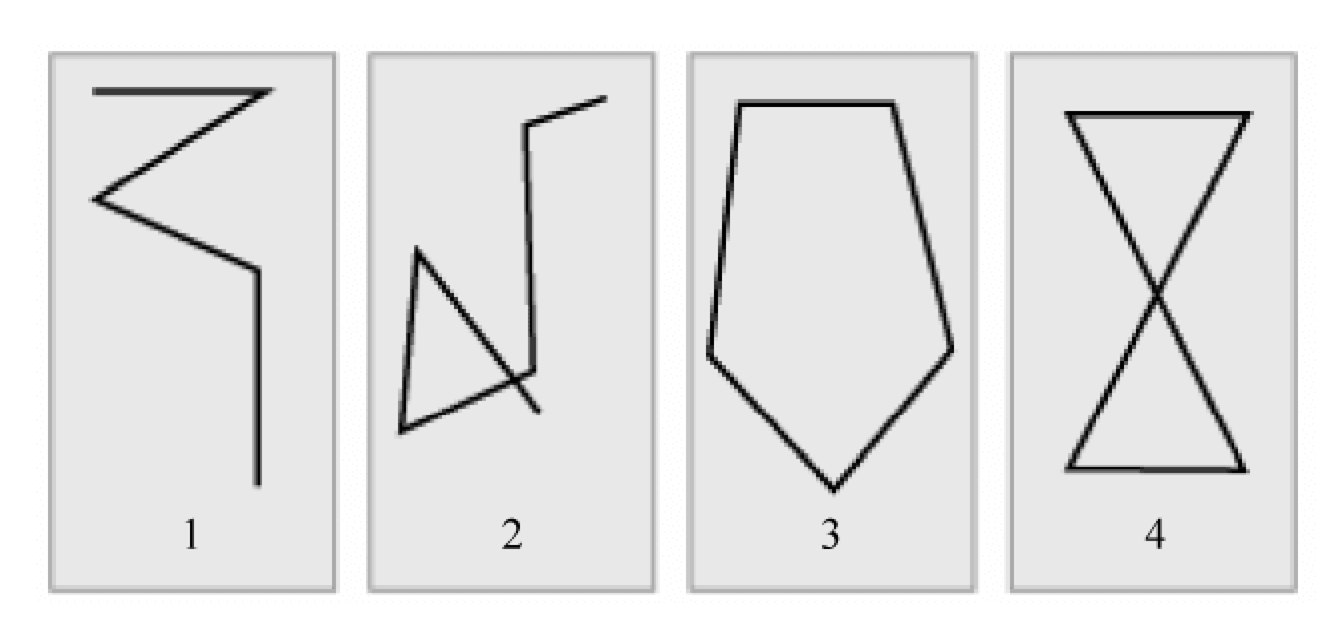


图 15.3　LineString 的简单和闭合

LineString 对象在 WKT 中使用关键字 LineString 表示，其格式为：

```
LineString(X1 Y1,X2 Y2[,X3 Y3]...)
```

LineString 中的每个点可以只使用 XY 表示，也可以使用 XYMZ 表示，每个点之间使用逗号分隔。

系统专门为构造 LineString 对象提供了以下方法：

- STLineFromText ('linestring_tagged_text' , SRID)；
- STLineFromWKB ('WKB_linestring' , SRID)。

例如要使用 WKT 构造一个 LineString 对象，对应的脚本如代码 15.8 所示。

代码 15.8　构造 LineString 对象

```
DECLARE @g geometry;
SET @g = geometry::STLineFromText('LINESTRING(0 0,1 1 2 3,3 4)',0);
```

在构造了 LineString 对象后，可以使用 ToString()等方法将对象实例的 WKT 输出，这个在介绍 Point 对象时已经讲解了，这里不再重复。

系统为 LineString 对象提供了很多个方法，用于判断或计算该对象的一些属性。

- STLength()返回 float 类型，表示实例的总长度。

例如，求一条线的长度的脚本如代码 15.9 所示。

代码 15.9　求 LineString 实例的总长度

```
DECLARE @g geometry;
SET @g = geometry::STLineFromText('LINESTRING(0 0,3 4)',0);
SELECT @g.STLength()    --获得线的长度
```

系统返回线的长度为 5。

- STStartPoint()返回一个 Point 类型，表示 LineString()实例的起始点。

LineString 的起始点为 WKT 中定义的第一个点，例如，要获得一个 LineString 对象的起始点的脚本如代码 15.10 所示。

代码 15.10　获得 LineString 实例的起始点

```
DECLARE @g geometry;
SET @g = geometry::STLineFromText('LINESTRING(0 0,3 4)',0);
SELECT @g.STStartPoint().ToString()    --获得线的起点
--系统返回结果：
POINT (0 0)
```

- STEndPoint()方法与 STStartPoint()方法类似，只是该方法返回的是 LineString 实例的最后一个点。
- STPointN（expression）返回一个 Point 类型，表示 LineString 实例中的第 *n* 个点，*n* 就是其传入的参数。

例如声明一个由 4 个点定义的 LineString 对象，获得该实例的第 2 个点的脚本如代码 15.11 所示。

代码 15.11　获得 LineString 实例中指定的点

```
DECLARE @g geometry;
SET @g = geometry::STLineFromText('LINESTRING(0 0,3 4,2 2,4 3)',0);
SELECT @g.STPointN(2).ToString()
--系统返回结果：
POINT (3 4)
```

注意：如果使用大于实例中点数的值来调用此方法，则返回 NULL。

- STNumPoints()方法返回 int 类型，表示 LineString 实例总点数。

该方法与 STNumGeometries()方法类似，只不过该方法用于计算 LineString 中的点，而 STNumGeometries()是用于计算一个集合中的对象数。

对于 LineString 中重复的点，则 STNumPoints()方法将会进行重复计数。例如创建一个

闭合的 LineString，计算其中的点数如代码 15.12 所示。

代码 15.12　获得 LineString 实例中的点数

```
DECLARE @g geometry;
SET @g = geometry::STLineFromText('LINESTRING(0 0,3 4,2 2,4 3,0 0)',0);
SELECT @g.STNumPoints()
--其中点(0,0)重复，但仍将计数，系统返回结果 5。
```

- STIsRing()返回一个 bit 类型，表示 LineString 实例是否是简单闭合的。

例如声明 2 个 LineString 实例，一个是简单闭合，一个是简单不闭合的，通过 STIsRing()方法即可判断这两个实例的区别，如代码 15.13 所示。

代码 15.13　判断 LineString 实例是否简单闭合的

```
DECLARE @g geometry;
SET @g = geometry::STLineFromText('LINESTRING(0 0,3 4,2 2,4 3,0 0)',0);
SELECT @g.STIsRing() --返回 1
SET @g = geometry::STLineFromText('LINESTRING(0 0,3 4,2 2,4 3)',0);
SELECT @g.STIsRing() --返回 0
```

- STIsClosed()和 STIsSimple()方法都返回 bit 类型，表示 LineString 实例是否是闭合和简单的。

显然，如果一个 LineString 实例的 STIsRing()方法返回 1，则 STIsClosed()和 STIsSimple()方法必然返回 1。

15.2.4　MultiLineString 线集的使用

MultiLineString 是零个或更多 geometry 或 geography LineString 实例的集合。如图 15.4 为 MultiLineString 的一些示例。

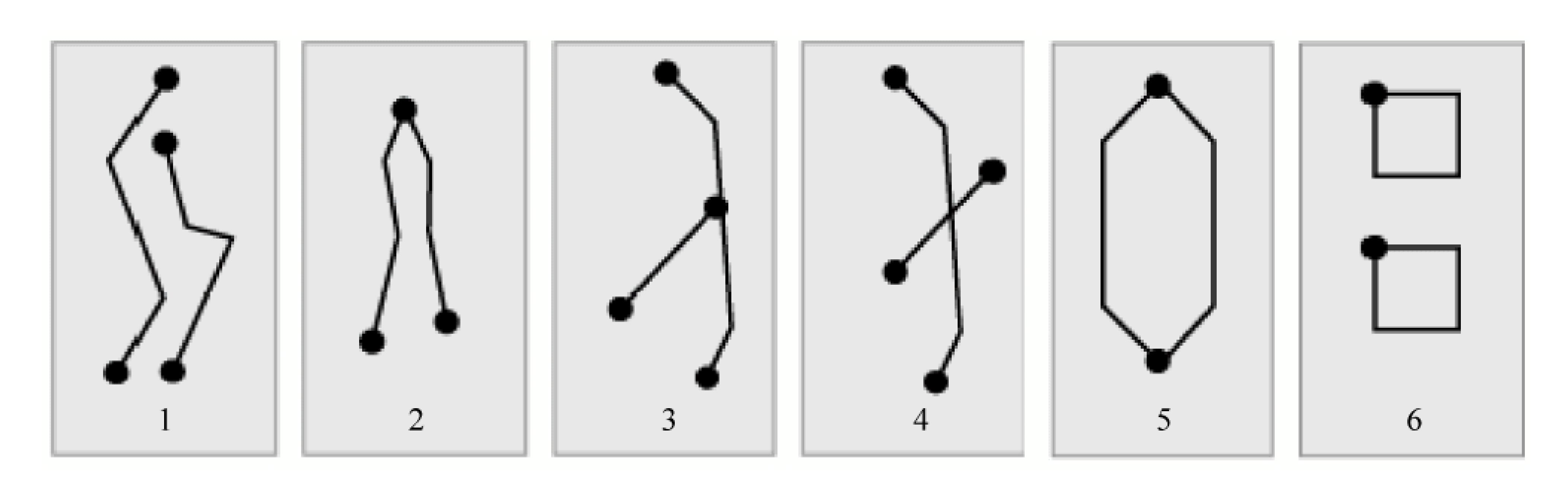


图 15.4　MultiLineString 示例

MultiLineString 实例仍然存在简单和闭合的概念，图 15.4 中：

- 图 1 显示的是一个简单的 MultiLineString 实例，其边界是其两个 LineString 元素的 4 个端点。
- 图 2 显示的是一个简单的 MultiLineString 实例，因为只有 LineString 元素的端点相交。边界是两个不重叠的端点。
- 图 3 显示的是一个不简单的 MultiLineString 实例，因为它的其中一个 LineString

元素的内部出现了相交。此 MultiLineString 实例的边界是 4 个端点。

- 图 4 显示的是一个不简单、非闭合的 MultiLineString 实例。
- 图 5 显示的是一个简单、非闭合的 MultiLineString。它没有闭合是因为它的 LineStrings 元素没有闭合。而其简单的原因在于，其任何 LineStrings 实例的内部都没有出现相交。
- 图 6 显示的是一个简单、闭合的 MultiLineString 实例。它是闭合的是因为它的所有元素都是闭合的。而其简单的原因在于，其所有元素都没有出现内部相交现象。

MultiLineString 对象在 WKT 中使用 MultiLineString 关键字表示，其语法格式为：

```
MultiLineString((X1 Y1,X2 Y2 [,X3 Y3]…),(Xm Ym,Xn Yn [,Xo Yo]) [,(…)]…)
```

MultiLineString 中使用括号将每个 LineString 括起来，然后使用逗号进行分隔。

系统为构造 MultiLineString 对象提供了以下方法：

- STMLineFromText ('multilinestring_tagged_text' , SRID)；
- STMLineFromWKB ('WKB_multilinestring' , SRID)。

例如要构造一个 MultiLineString 实例，其脚本如代码 15.14 所示。

代码 15.14　构造 MultiLineString 实例

```
DECLARE @g geometry;
SET @g = geometry::STMLineFromText(
'MULTILINESTRING((100 100, 200 200), (3 4, 7 8, 10 10))', 0);
```

在 LineString 中适用的方法在 MultiLineString 中大部分仍然适用。例如 STLength()方法将返回 MultiLineString 中所有 LineString 对象的长度总和。STIsClosed()和 STIsSimple()方法分别用于判断 MultiLineString 是否是闭合和简单的。

15.2.5　Polygon 面的使用

Polygon 是存储为一系列点的二维表面，这些点定义一个外部边界环和零个或多个内部环。可以从至少具有 3 个不同点的环中构建一个 Polygon 实例。Polygon 实例也可以为空。

Polygon 的外部环和任意内部环定义了其边界。环内部的空间定义了 Polygon 的内部。Polygon 的内部环在单个切点处既可与自身接触，也可彼此接触，但如果 Polygon 的内部环交叉，则该实例无效。例如图 15.5 所示为 Polygon 的示例。

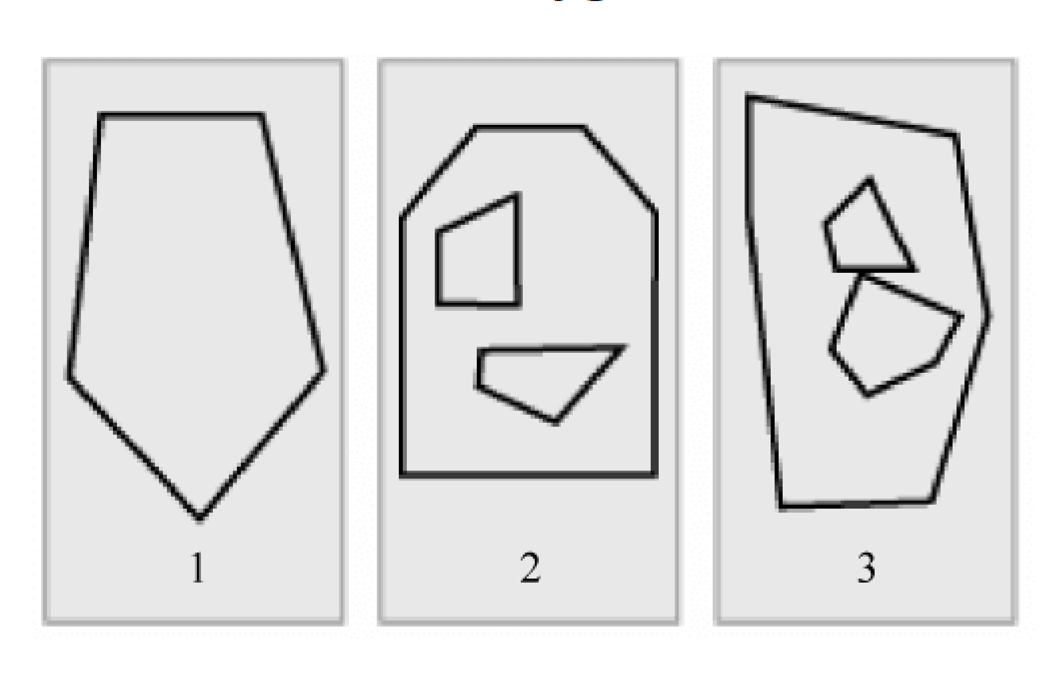


图 15.5　Polygon 的示例

在图 15.5 中 3 个多边形都是有效的 Polygon 实例。

- 图 1 是由外部环定义其边界的 Polygon 实例。
- 图 2 是由外部环和两个内部环定义其边界的 Polygon 实例。内部环内的面积是 Polygon 实例外部环的一部分。
- 图 3 是一个有效的 Polygon 实例，因为其内部环在单个切点处相交。

Polygon 在 WKT 中使用 Polygon 关键字表示，其语法为：

```
Polygon(Ring1 [,Ring2]...)
```

其中 Ring1 是该多边形的外环，使用 LineString 的格式为：

```
(X1 Y1 [,X2 Y2]…,X1 Y1)
```

LineString 第一个点和最后一个点必须重合形成环。

如果 Polygon 实例中有内部环，则使用 Ring2、Ring3 等来表示。内部环必须在外环的内部或者单切点相交，否则将会是一个无效的 Polygon。

例如要构造一个简单的没有内部环的正方形，则对应的脚本如代码 15.15 所示。

代码 15.15　构造正方形的 Polygon 实例

```
DECLARE @g geometry;
SET @g = geometry::Parse('POLYGON((1 0, 0 1, 1 2, 2 1, 1 0))');
```

如果要构造一个具有内部环的 Polygon 对象，则对应脚本如代码 15.16 所示。

代码 15.16　构造有内部环的 Polygon 实例

```
DECLARE @g geometry;
SET @g = geometry::Parse('POLYGON((1 0, 0 1, 1 2, 2 1, 1 0),(0.5 0.6,0.7
0.7,0.5 0.7,0.5 0.6))');
```

系统为 Polygon 对象提供很多方法，便于获得实例的属性。

- STArea()方法返回 float 类型，用于计算 Polygon 实例的面积。

例如构造一个 Polygon 对象，计算其面积的脚本如代码 15.17 所示。

代码 15.17　获得 Polygon 实例的面积

```
DECLARE @g geometry;
SET @g = geometry::Parse('POLYGON((1 0, 0 1, 1 2, 2 1, 1 0))');
SELECT @g.STArea()
--系统返回该多边形的面积：2
```

注意：对于 Point、LineString 等对象，使用 STArea()方法将返回结果 0。

- STExteriorRing()方法返回一个 LineString 类型，表示 Polygon 实例的外环。

例如构造一个回字形的 Polygon 对象，获得其外环的脚本如代码 15.18 所示。

代码 15.18　获得 Polygon 实例的外环

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 3 0, 3 3, 0 3, 0 0),(2 2,
2 1, 1 1, 1 2, 2 2))', 0);
SELECT @g.STExteriorRing().ToString();
--系统返回结果：
LINESTRING (0 0, 3 0, 3 3, 0 3, 0 0)
```

- STNumInteriorRing()方法返回 int 型数据，表示 Polygon 实例的内环数。

例如构造一个具有内环的 Polygon 实例，获得其内环数的脚本如代码 15.19 所示。

代码 15.19　获得 Polygon 实例的内环数

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 3 0, 3 3, 0 3, 0 0),(2 2,
2 1, 1 1, 1 2, 2 2))', 0);
SELECT @g.STNumInteriorRing()
--系统返回结果：1
```

- STInteriorRingN(expression)方法返回 LineString 对象，表示指定的 Polygon 中的第 *n* 个内环。数值 *n* 正是该方法的参数。

例如，获得一个 Polygon 实例的第 1 个内环的脚本，如代码 15.20 所示。

代码 15.20　获得 Polygon 实例的一个内环

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 3 0, 3 3, 0 3, 0 0),(2 2,
2 1, 1 1, 1 2, 2 2))', 0);
SELECT @g.STInteriorRingN(1).ToString();
--系统返回结果：
LINESTRING (2 2, 2 1, 1 1, 1 2, 2 2)
```

注意：如果传入的参数大于实际的环数，则系统将返回 NULL。

- STCentroid()方法返回 Point 类型，表示 Polygon 实例的几何中心，即重心。

例如构造一个 Polygon 对象实例，获得其重心的脚本如代码 15.21 所示。

代码 15.21　获得 Polygon 实例的几何中心

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 3 0, 3 3, 0 3, 0 0),(2 2,
2 1, 1 1, 1 2, 2 2))', 0);
SELECT @g.STCentroid().ToString(); --获得几何中心
--系统返回结果：
POINT (1.5 1.5)
```

- STPointOnSurface()方法返回 Point 类型，表示位于 geometry 实例内部的任意点。

例如构造一个 Polygon 实例，获得其中一个点的脚本如代码 15.22 所示。

代码 15.22　获得 Polygon 实例中的一个点

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 3 0, 3 3, 0 3, 0 0),(2 2,
2 1, 1 1, 1 2, 2 2))', 0);
SELECT @g.STPointOnSurface().ToString();
--系统返回该多边形中的一个点：
POINT (1.6666666666666667 2.6666666666666665)
```

15.2.6　MultiPolygon 面集的使用

MultiPolygon 实例是零个或更多个 Polygon 实例的集合。例如图 15.6 所示为

MuliPolygon 对象的示例。

图 15.6 中都是合法的 MuliPolygon 示例。

- ❑ 图 1 是一个包含两个 Polygon 元素的 MultiPolygon 实例。边界由两个外环和三个内环界定。
- ❑ 图 2 是一个包含两个 Polygon 元素的 MultiPolygon 实例。边界由两个外环和三个内环界定。这两个 Polygon 元素在切点处相交。

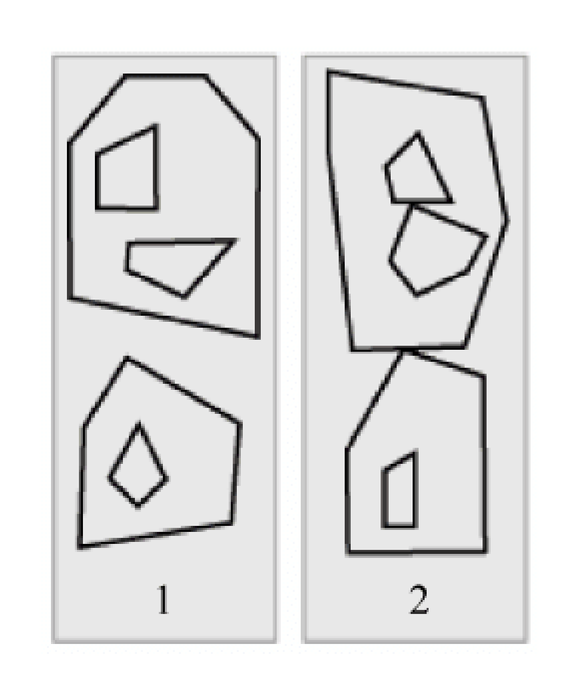


图 15.6　MuliPolygon 的示例

MuliPolygon 对象在 WKT 中使用 MuliPolygon 关键字表示，其语法格式为：

```
MuliPolygon( Polygon1 [,Polygon2]...)
```

其中的 Polygon1、Polygon2 使用括号括起来的多边形表示。

系统为构造 MuliPolygon 提供了以下方法。

- ❑ STMPointFromText ('multipoint_tagged_text' , SRID)；
- ❑ STMPolyFromWKB ('WKB_multipolygon' , SRID)。

例如要构造一个 MuliPolygon 实例，该实例中包含了两个多边形，则对应脚本如代码 15.23 所示。

代码 15.23　构造 MuliPolygon 实例

```
DECLARE @g geometry;
SET @g = geometry::STMPolyFromText('MULTIPOLYGON(((47.653 -122.358, 47.649
-122.348, 47.658 -122.358, 47.653 -122.358)), ((47.656 -122.341, 47.661
-122.341, 47.661 -122.351, 47.656 -122.341)))', 0);
```

对于 Polygon 实例使用的方法，大多数也可以在 MuliPolygon 实例中使用。例如可以使用 STArea()方法获得所有多边形的总面积和，STCentroid()方法获得多个多边形共同形成的几何中心，STPointOnSurface()方法获得其中任意一个多边形中的任意一个点。

15.2.7　GeometryCollection 几何集合的使用

GeometryCollection 是零个或更多个 geometry 或 geography 实例的集合。GeometryCollection 可以为空。GeometryCollection 对象中可以包含前面提到的所有 Geometry 类型，GeometryCollection 甚至还可以包含另外的 GeometryCollection 对象。在 WKT 中使用 GeometryCollection 关键字来表示 GeometryCollection 对象，其语法格式为：

```
GeometryCollection(Geometry1 [,Geometry2]…)
```

其中的 Geometry1、Geometry2 等是其他 Geometry 对象的 WKT 表达式，每个 Geometry 对象之间使用逗号分隔。

系统为构造 GeometryCollection 实例提供了以下方法。

- ❑ STGeomCollFromText ('geometrycollection_tagged_text'，SRID)；
- ❑ STGeomCollFromWKB ('WKB_geometrycollection'，SRID)。

例如要创建一个 Point 对象和一个 Polygon 对象的集合，则对应的脚本如代码 15.24

所示。

代码 15.24　构造 GeometryCollection 对象实例

```
DECLARE @g geometry;
SET @g = geometry::STGeomCollFromText( --构造几何集合对象
'GEOMETRYCOLLECTION(POINT(3 3 1), POLYGON((0 0 2, 1 10 3, 1 0 4, 0 0 2)))',
1);
```

GeometryCollection 中可以包含 GeometryCollection 对象，例如创建一个 Geometry-Collection 对象，该对象将一个 LineString 对象和一个 GeometryCollection 对象包含在其中，如代码 15.25 所示。

代码 15.25　GeometryCollection 中包含 GeometryCollection 对象

```
DECLARE @g geometry;
SET @g = geometry::STGeomCollFromText('GEOMETRYCOLLECTION(LINESTRING(1 2,
3 4),GEOMETRYCOLLECTION(POINT(3 3 1), POLYGON((0 0 2, 1 10 3, 1 0 4, 0 0
2))))', 1);
```

在前面提到的 STNumGeometries()和 STGeometryN()方法适用于 GeometryCollection 对象实例。例如获得一个 GeometryCollection 实例中包含的 Geometry 对象数目和取出第 1 个 Geometry 对象的脚本如代码 15.26 所示。

代码 15.26　获得 GeometryCollection 中包含的 Geometry 对象数

```
DECLARE @g geometry;
SET @g = geometry::STGeomCollFromText('GEOMETRYCOLLECTION(LINESTRING(1 2,
3 4),GEOMETRYCOLLECTION(POINT(3 3 1), POLYGON((0 0 2, 1 10 3, 1 0 4, 0 0
2))))', 1);
SELECT @g.STNumGeometries ()           --系统返回：2
SELECT @g.STGeometryN(1).ToString()    --系统返回：LINESTRING (1 2, 3 4)
```

注意：STNumGeometries()返回的是直属于 GeometryCollection 实例的对象数目，而不是通过递归获得其所有最小 Geometry 单元的数目。

15.2.8　操作几何图形实例

就像 Polygon 对象实例的 STExteriorRing()方法返回一个 LineString 对象一样，SQL Server 提供了多个通过一个对象实例创建另外一个对象实例的方法。

1．STBuffer()方法获得缓冲区

STBuffer(distance)方法返回一个几何对象，该对象表示所有与 geometry 实例的距离小于或等于指定值的点的并集（又称为缓冲区）。参数 distance 就是指定的距离。例如对于一个点来说，到该点的距离小于等于指定值的点的并集就组成了一个圆，该圆的半径等于指定的值。SQL Server 中并没有提供圆或者弧的对象，圆用 Polygon 对象来表示，弧则使用 LineString 来表示。例如求距离点（0,0）长度为 1 的缓冲区的 SQL 脚本如代码 15.27 所示。

代码 15.27　求点的缓冲区

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('Point(0 0)', 0);
SELECT @g.STBuffer(1).ToString();        --距离为 1 的缓冲区的 WKT
```

系统将返回以（0，0）为圆心，1 为半径的圆对应的 Polygon 对象，如下所示。

```
POLYGON ((0 -1, 0.051459848880767822 -0.99869883060455322,
0.10224419832229614 -0.99483710527420044, 0.15229016542434692
-0.98847776651382446, 0.20153486728668213 -0.97968357801437378,
0.24991559982299805 -0.96851736307144165,
...
,0 -1))
```

注意：由于 Polygon 表示的是多边形，所以用 Polygon 对象表示圆，只有使用正几十边形或正几百边形的临近法来表示。

再例如要求一个折线的缓冲区，其对应的脚本如代码 15.28 所示。

代码 15.28　求折线的缓冲区

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('LineString(0 2,0 0,2 0)', 0);
SELECT @g.STBuffer(1).ToString();        --折线的缓冲区图形的 WKT
```

对于一个直角折线，其距离为 1 的缓冲区如图 15.7 所示。如果将缓冲区距离设置为 2、3、4 时，则对应的缓冲区图形如图 15.8 所示。



图 15.7　折线的缓冲区　　　　图 15.8　折线的不同距离的缓冲区

2. BufferWithTolerance()方法获得指定公差的缓冲区

BufferWithTolerance（distance，tolerance，relative）方法返回表示所有点值的并集的几何对象，这些点到 geometry 实例的距离小于或等于指定值，允许存在指定的公差。

Distance 是一个指定到 geometry 实例距离的 float 表达式，缓冲区就是环绕该实例而计算出的。tolerance 指定缓冲区距离的公差 float 表达式。“公差”指的是理想的缓冲区距离与返回的线性近似段之间的最大偏差。例如，点的理想缓冲区距离为圆圈，但是这必须与多边形近似。公差越小，多边形具有的点就越多，这将增加结果的复杂度，但可减少错误。

relative 一个 bit，指定 tolerance 值是相对值还是绝对值。如果为“TRUE”或 1，则公差为相对值并按 tolerance 参数，与该实例的边界框直径之间的乘积进行计算。如果为“FALSE”或 0，则公差为绝对值并且 tolerance 值为理想缓冲区距离与返回的线性近似段之

间的绝对最大偏差。例如求点（3，3）的距离为 1 的缓冲区，公差为 0.5 的绝对值，则对应的脚本如代码 15.29 所示。

代码 15.29　求点的有公差的缓冲区

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POINT(3 3)', 0);
SELECT @g.BufferWithTolerance(1, .5, 0).ToString();  --公差缓冲区图形的 WKT
```

不同的公差值返回不同的多边形，对于 0.5、0.2 和 0.05 不同的公差值，对应的缓冲区图，如图 15.9 所示。



图 15.9　不同的公差值对应的缓冲区

3．Reduce()方法获得近似值

Reduce（tolerance）方法返回给定 geometry 实例的近似值，该值通过对实例运行具有给定公差的 Douglas-Peucker 算法来生成。其参数 tolerance 类型为 float 的值。tolerance 是输入到 Douglas-Peucker 算法的公差。对于集合类型，此算法单独作用于包含在该实例中的每个 geometry。此算法不修改 Point 实例。

在 LineString 实例上，Douglas-Peucker 算法保持该实例的原始起点和终点，并以迭代方式重新添加原始实例中与结果偏差最大的点，直到任何点的偏差都不超出给定公差。

在 Polygon 实例上，Douglas-Peucker 算法独立应用于每个环。如果返回的 Polygon 实例无效，该方法将生成 FormatException。例如，如果应用 Reduce()方法的目的在于简化实例中的每个环，而且所生成的环发生重叠，则会创建无效的 MultiPolygon 实例。例如对于一个 LineString 实例，返回其近似值的脚本如代码 15.30 所示。

代码 15.30　返回 LineString 实例的近似值

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 0, 0 1, 1 0, 2 1, 3 0, 4
1)', 0);
SELECT @g.Reduce(.75).ToString();  --线的近似值
--系统返回结果:
LINESTRING (0 0, 0 1, 3 0, 4 1)
```

4．STConvexHull()方法获得凸包对象

STConvexHull()方法返回表示 geometry 实例的凸包的对象。Point 或共线 LineString 实例将生成与该输入具有相同类型的实例。STConvexHull()方法主要用于 Polygon 对象和 GeometryCollection 对象。例如对于一个凹多边形，获得该多边形实例的凸包的脚本如代码 15.31 所示。

代码 15.31　返回 Polygon 对象的凸包

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 0 2, 1 1, 2 2, 2 0, 0 0))',
0);
SELECT @g.STConvexHull().ToString();   --凸包图形的 WKT
--系统返回结果:
POLYGON ((2 0, 2 2, 0 2, 0 0, 2 0))
```

对应的凹多边形及其凸包如图 15.10 所示。

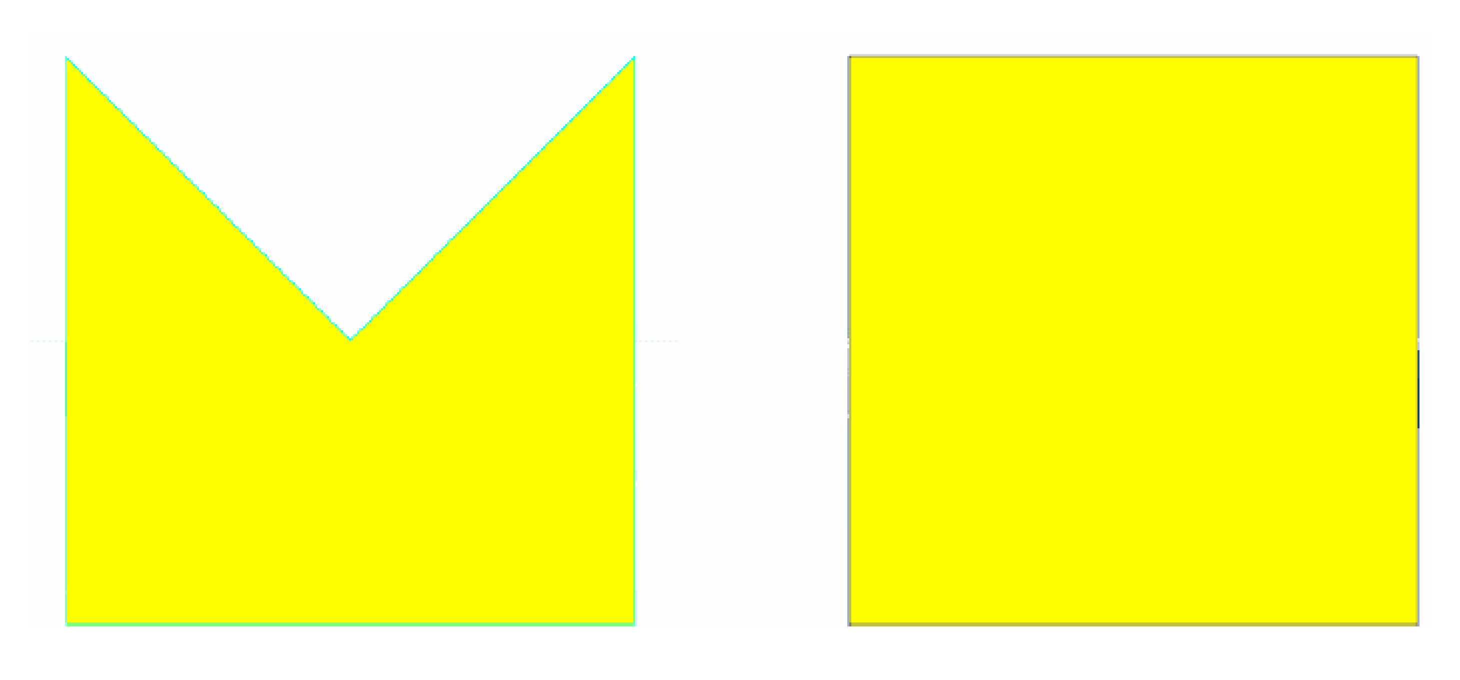


图 15.10　多边形和其凸包

5．STIntersection()方法获得两实例交点

STIntersection（other_geometry）方法返回一个对象，该对象表示一个 geometry 实例与另一个 geometry 实例的交点。其中参数 other_geometry 是要与调用 STIntersection()方法所在的实例进行比较的另一个 geometry 实例，进行比较的目的是确定这两个实例是否相交。

注意：如果 geometry 实例的空间引用 ID（SRID）不匹配，则 STIntersection()方法始终返回 Null。

如果两个实例不相交，则返回空的 GeometryCollection 对象。例如有两个 Polygon 对象实例，获得这两个实例的公共部分的脚本如代码 15.32 所示。

代码 15.32　获得两个 Polygon 实例的公共部分

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 0 2, 2 2, 2 0, 0 0))', 0);
SET @h = geometry::STGeomFromText('POLYGON((1 1, 3 1, 3 3, 1 3, 1 1))', 0);
SELECT @g.STIntersection(@h).ToString();   --相交部分的 WKT
--系统返回结果:
POLYGON ((1 1, 2 1, 2 2, 1 2, 1 1))
```

6．STUnion()方法获得两实例并集

STUnion（other_geometry）方法返回一个对象，该对象表示两个 geometry 实例的并集。其参数 other_geometry 表示与调用 STUnion()方法所在的实例形成一个并集的另一个 geometry 实例。

点与点的并集将组成 MultiPoint 对象，线与线的并集将组成新的 LineString 对象或者

MultiLineString 对象，多边形与多边形的并集如果能够组成一个新的多边形则组成 Polygon 对象，否则组成 MultiPolygon 对象，其他类型对象的并集将返回 GeometryCollection 对象。例如有两个 Polygon 实例，这两个实例相交，获得这两个实例的并集的脚本如代码 15.33 所示。

代码 15.33　获得两个 Polygon 实例的并集

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 0 2, 2 2, 2 0, 0 0))', 0);
SET @h = geometry::STGeomFromText('POLYGON((1 1, 3 1, 3 3, 1 3, 1 1))', 0);
SELECT @g.STUnion(@h).ToString();  --并集的 WKT
--系统返回结果：
POLYGON ((0 0, 2 0, 2 1, 3 1, 3 3, 1 3, 1 2, 0 2, 0 0))
```

再如两个 LineString 实例，获得这两个实例的并集的脚本如代码 15.34 所示。

代码 15.34　获得两个 LineString 实例的并集

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('LineString(0 0, 0 2, 2 2, 2 0)', 0);
SET @h = geometry::STGeomFromText('LineString(0 0, 3 1, 3 3, 1 3, 3 0)',
0);
SELECT @g.STUnion(@h).ToString();  --两条线的并集
--系统返回结果：
LINESTRING (3 0, 2.4545454545454546 0.81818181818181834, 2
0.66666666666666674, 2 1.5, 2.4545454545454546 0.81818181818181834, 3 1,
3 3, 1 3, 1.6666666666666665 2, 2 2, 2 1.5, 1.6666666666666665 2, 0 2, 0
0, 2 0.66666666666666674, 2 0)
```

7. STDifference()方法获得在一个实例存在而在另一实例不存在的对象

STDifference（other_geometry）方法返回一个对象，该对象表示来自一个 geometry 实例的点，这些点在另一个 geometry 实例中不存在。参数 other_geometry 是另一个 geometry 实例，指示要从调用了 STDifference()的实例中删除哪些点。

注意：STDifference()方法一般用于两个相同类型的 Geometry 实例中，对于从一个 Polygon 对象中删除一个 LineString 或 Point 对象并没有多大意义。

例如有一个矩形 Polygon 实例，再给出一小矩形 Polygon 实例，获得这两个 Polygon 实例中在第 1 个实例中存在而在第 2 个实例中不存在的点的集合，其脚本如代码 15.35 所示。

代码 15.35　获得两个 Polygon 实例的 1 个中不重叠的部分

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 0 2, 2 2, 2 0, 0 0))', 0);
SET @h = geometry::STGeomFromText('POLYGON((1 1, 3 1, 3 3, 1 3, 1 1))', 0);
SELECT @g.STDifference(@h).ToString(); --不重叠部分的 WKT
--系统返回结果：
POLYGON ((0 0, 2 0, 2 1, 1 1, 1 2, 0 2, 0 0))
```

注意：如果两个实例不相交，则 STDifference()方法返回调用该方法的实例本身；如果两实例相交，而调用该方法的实例维度较低，则返回空的 GeometryCollection 对象。

8．STSymDifference()方法获得非交集对象

STSymDifference（other_geometry）返回一个对象，该对象表示符合如下条件的所有点：位于一个 geometry 实例或者另一个 geometry 实例中，但不同时位于这两个实例中。其参数 other_geometry 是另一个 geometry 实例，即除调用 STSymDistance()方法所在的实例以外的另一个实例。例如对于两个 Polygon 实例，获得其不相交部分点的集合的脚本如代码 15.36 所示。

代码 15.36　获得两个 Polygon 实例不相交部分

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 0 2, 2 2, 2 0, 0 0))', 0);
SET @h = geometry::STGeomFromText('POLYGON((1 1, 3 1, 3 3, 1 3, 1 1))', 0);
SELECT @g.STSymDifference(@h).ToString();  --不相交部分的 WKT
--系统返回结果：
MULTIPOLYGON (((2 1, 3 1, 3 3, 1 3, 1 2, 2 2, 2 1)), ((0 0, 2 0, 2 1, 1 1,
1 2, 0 2, 0 0)))
```

如果两个实例的类型不同，例如一个 LineString 实例和一个 Polygon 实例，若 LineString 实例并没有被 Polygon 实例完全覆盖，则不相交部分返回 GeometryCollection 对象。若 LineString 实例被 Polygon 实例完全覆盖，则不相交部分返回 Polygon 实例本身。获得 LineString 实例和 Polygon 实例的不相交部分的脚本如代码 15.37 所示。

代码 15.37　获得 1 个 LineString 和 1 个 Polygon 不相交部分

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 0 2, 2 2, 2 0, 0 0))', 0);
SET @h = geometry::STGeomFromText('LineString(1 1, 1 1.5)', 0);
SELECT @g.STSymDifference(@h).ToString();  --不相交部分的 WKT
--系统将返回结果：
POLYGON ((0 0, 2 0, 2 2, 0 2, 0 0))
```

9．STGeometryType()方法获得实例名称

STGeometryType()方法用于返回由 geometry 实例表示的开放地理空间联盟（OGC）类型名称。由 STGeometryType()方法返回的 OGC 类型包括 Point、LineString、Polygon、GeometryCollection、MultiPoint、MultiLineString 和 MultiPolygon。例如获得一个 Polygon 实例名称的脚本如代码 15.38 所示。

代码 15.38　获得实例的名称

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 3 0, 3 3, 0 3, 0 0))', 0);
SELECT @g.STGeometryType();  --获得实例名称
--系统将返回结果：Polygon
```

10．InstanceOf()方法判断实例类型

InstanceOf（geometry_type）是用于测试 geometry 实例是否与指定的类型相同的方法。如果 geometry 实例的类型与指定的类型相同，或者指定的类型是该实例类型的父类型，则返回 1；否则，返回 0。

其参数 geometry_type 是一个 nvarchar(4000)字符串，必须为以下之一：Geometry、Point、Curve、LineString、Surface、Polygon、GeometryCollection、MultiSurface、MultiPolygon、MultiCurve、MultiLineString 和 MultiPoint。如果将任何其他字符串用于输入，此方法将引发 ArgumentException。例如判断一个 MultiPoint 实例是否是 GeometryCollection 对象，则对应的脚本如代码 15.39 所示。

代码 15.39　判断实例是否为给定的类型

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('MULTIPOINT(0 0, 13.5 2, 7 19)', 0);
SELECT @g.InstanceOf('GEOMETRYCOLLECTION');    --判断是否为几何集合
--系统返回结果：1
```

11．STIsValid()方法判断实例格式是否正确

STIsValid()方法用于判断 geometry 实例的格式是否符合 OGC 类型，如果可确定该实例的格式正确，则返回 1。如果 geometry 实例格式不正确，则返回 0。例如定义一个无效的 LineString 实例，判断该实例格式是否正确的脚本如代码 15.40 所示。

代码 15.40　判断实例格式是否正确

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 0, 2 2, 1 1)', 0);
SELECT @g.STIsValid();  --验证 WKT 描述是否正确
--系统返回结果：0
```

12．MakeValid()方法获得有效格式的实例

MakeValid()方法用于将无效 geometry 实例转换为具有有效开放地理空间联盟（OGC）类型的 geometry 实例。

注意：此方法可能会导致 geometry 实例的类型有所变化，还会导致 geometry 实例的点略微移位。

例如前面提到的 LineString 实例，获得其有效实例的脚本如代码 15.41 所示。

代码 15.41　获得实例的有效格式

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 0, 2 2, 1 1)', 0);
SELECT @g.MakeValid().ToString();  --获得有效 WKT
--系统返回结果：
LINESTRING (2 2, 1 1, 0 0)
```

15.2.9　几何图形实例的属性和方法

几何图形在实例化后可以调用其属性和方法，前面在介绍每一个 Geometry 类型时都介绍了一些该类型可使用的属性和方法，本节再集中统一分类介绍一些实例可使用的属性和方法。

1．点数

所有非空 geometry 实例都由“点”组成。这些点表示在其上绘制几何图形的面的 X 和 Y 坐标。geometry 提供许多用于查询实例的点的内置方法。

- STNumPoints()（geometry 数据类型）返回构成实例的点数。
- STPointN()返回实例中的特定点。
- STPointOnSurface()返回位于实例上的某个任意点。
- STStartPoint()返回实例的起始点。
- STEndpoint()返回实例的终点。
- STX()（geometry 数据类型）返回点实例的 X 坐标。
- STY()返回点实例的 Y 坐标。
- STCentroid()返回多边形实例的几何中心点。

这些属性和方法在前面介绍具体 Geometry 类型时已经介绍和举例说明了，这里不再重复。

2．维度

非空 geometry 实例可以为零维、一维或二维。零维 geometries（例如 Point 和 MultiPoint）没有长度或面积。一维对象（例如 LineString 和 MultiLineString）具有长度。二维实例（例如 Polygon 和 MultiPolygon）具有面积和长度。空实例将报告为–1 维，并且 GeometryCollection 将根据其内容类型报告一个面积。与维度相关的方法有：

- STDimension()返回实例的维度。
- STLength()返回实例的长度。
- STArea()返回实例的面积。

其中 STLength()和 STArea()在前面已经使用过，这里主要讲 STDimension()方法。

STDimension()方法返回 geometry 实例的最大维度。空实例调用该方法将返回–1。对于一个 GeometryCollection 对象，如果其中包含了 LineString 或 MultiLineString，而没有包含 Polygon 和 MultiPolygon 对象，则 STDimension()方法返回 1，如果只包含 Point 和 MultiPoint 对象，则其返回值为 0。例如，获得一个包含多个对象的 GeometryCollection 对象的维度脚本，如代码 15.42 所示。

代码 15.42　获得维度

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('GeometryCollection(POLYGON((0 0, 0 2,
2 2, 2 0, 0 0)),LineString(1 1, 1 1.5),Point(0 1))', 0);
SELECT @g.STDimension();   --返回维度
--系统返回结果：2
```

注意： 如果 GeometryCollection 实例中包含了 MultiPolygon 对象，不一定其维度就是 2，因为 MultiPolygon 对象为空时维度为–1。

3．空

“空”geometry 实例不包含任何点。空的 LineString 和 MultiLineString 实例的长度为 0。空的 Polygon 和 MultiPolygon 实例的面积为 0。系统提供了方法 STIsEmpty()用于判断实例是否为空，如果为空，则返回 1；否则返回 0。空对象在 WKT 中使用 EMPTY 关键字，例如：

```
POINT EMPTY      --定义空的几何实例
MULTIPOLYGON EMPTY
POLYGON EMPTY
```

创建一个空的多边形实例，使用该方法判断实例是否为空的脚本如代码 15.43 所示。

代码 15.43　判断实例是否为空

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON EMPTY', 0);
SELECT @g.STIsEmpty(); --是否为空
--系统返回结果：1
```

4．简单与闭合

在前面介绍 LineString 对象时提到了简单与闭合。简单的对象必须符合以下两个要求。

- ❑ 实例的每个图形不能与自身相交，但其终点除外。
- ❑ 实例的任何两个图形可在某个点上相交，但两个边界上的点除外。

“闭合的”geometry 实例是指起始点和终点相同的图形。Polygon 实例是闭合的。Point 实例不是闭合的。环是一个简单、闭合的 LineString 实例。

关于简单与闭合，系统提供了以下方法：

- ❑ STIsSimple()确定实例是否是简单的。
- ❑ STIsClosed()确定实例是否闭合。
- ❑ STIsRing()确定实例是否为环。
- ❑ STExteriorRing()返回多边形实例的外环。
- ❑ STNumInteriorRing()返回多边形的内环数。
- ❑ STInteriorRingN()返回多边形的指定内环。

5．边界、内部和外部

geometry 实例的“内部”是指由实例占用的空间，而“外部”是指未占用的空间。“边界”由 OGC 定义，如下所示。

- ❑ Point 和 MultiPoint 实例没有边界。
- ❑ LineString 和 MultiLineString 边界由起始点和终点形成，并删除那些出现次数为偶数的点。

STBoundary()方法返回实例的边界。例如对于一个 Polygon 对象，获得其边界的脚本如代码 15.44 所示。

代码 15.44　获得 Polygon 对象的边界

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 0 2, 2 2, 2 0, 0 0),(1 1,
1 1.5,1.5 1.5 1.5 1,1 1))', 0);
SELECT @g.STBoundary().ToString(); --对象边界 WKT
--系统返回结果：
MULTILINESTRING ((1 1, 1 1.5, 1.5 1.5, 1 1), (0 0, 2 0, 2 2, 0 2, 0 0))
```

从该示例可以看出，Polygon 对象的边界就是其外环和内环的并集。

注意：如果 LineString 是一个环，则其边界为空的 GeometryCollection 对象。

6. 包络线

geometry 实例的“包络线”又称为“边界框”，它是一个由实例的最小和最大坐标（X，Y）形成的轴对齐矩形。系统提供 STEnvelope()方法用于返回实例的包络线。由包络线的定义可知，该方法返回的是一个图形，比如是一个实心的 Polygon 矩形实例。例如对于一个 LineString 实例，获得其包络线的脚本如代码 15.45 所示。

代码 15.45　获得 LineString 实例的包络线

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 0, 2 3)', 0);
SELECT @g.STEnvelope().ToString(); --对象包络线 WKT
--系统返回结果：
POLYGON ((0 0, 2 0, 2 3, 0 3, 0 0))
```

7. 空间引用标识符

空间引用标识符被简写为 SRID，是指定 geometry 实例所在的坐标系的标识符。两个拥有不同 SRID 的实例是不可比的。

系统为实例提供了可供读写的属性 STSrid，该属性为 int 型。在构造 Geometry 对象实例时可以指定 SRID，也可以在构造后通过 STSrid 属性修改实例的 SRID。例如要获得一个 Polygon 对象的 SRID 和修改 SRID 如代码 15.46 所示。

代码 15.46　获得和修改 SRID

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 3 0, 3 3, 0 3, 0 0))', 1);
SELECT @g.STSrid; --返回
SET @g.STSrid = 3;
SELECT @g.STSrid; --返回
```

15.2.10　几何图形实例之间的关系

对于同一个 SRID 下的多个实例，它们之间是否存在着相交、接触、重叠、包含等关系，系统都提供了许多内置方法以获得多个实例之间的关系。

1. STEquals()方法确定两个实例是否包含相同的点集

STEquals（other_geometry）方法表示如果一个 geometry 实例表示的点集与另一个

geometry 实例相同，则返回 1；否则，返回 0。参数 other_geometry 是另一个 geometry 实例，将与在其上调用 STEquals()方法的实例进行比较。

例如对于一个 LineString 和 MultiLineString 实例，MuliLineString 实例中由 2 条 LineString 组成，判断这两个实例中的点集是否相同的脚本如代码 15.47 所示。

代码 15.47　判断 2 个实例的点集是否相同

```
DECLARE @g geometry
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 2, 2 0, 4 2)', 0);
SET @h = geometry::STGeomFromText('MULTILINESTRING((4 2, 2 0), (0 2, 2 0))',
0);
SELECT @g.STEquals(@h);  --是否相同
--系统返回结果：1
```

2. STDisjoint()方法确定两个实例是否不相接

STDisjoint（other_geometry）方法表示如果一个 geometry 实例与另一个 geometry 实例在空间上不相联，则返回 1；否则，返回 0。即如果两个 geometry 实例的点集不存在共同的部分，则这两个实例不相联。

例如给出一个 LineString 实例和一个 Point 实例，Point 实例在 LineString 实例上，则判断它们是否不相接的脚本如代码 15.48 所示。

代码 15.48　判断 2 个实例是否不相接

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 2, 2 0, 4 2)', 0);
SET @h = geometry::STGeomFromText('POINT(1 1)', 0);
SELECT @g.STDisjoint(@h);   --是否不相接
--由于点在线上，所以返回结果：0
```

3. STIntersects()方法确定两个实例是否相交

STIntersects（other_geometry）方法表示如果 geometry 实例与另一个 geometry 实例相交，则返回 1；否则，返回 0。该函数与 STDisjoint()函数的作用正好相反，如果两个 geometry 实例中的点集存在共同的部分，则这两个实例就相交。例如两个 LineString 实例，它们的图形存在交点，则判断其是否相交的脚本如代码 15.49 所示。

代码 15.49　判断 2 个实例是否相交

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 2, 2 0)', 0);
SET @h = geometry::STGeomFromText('LINESTRING(2 2,0 0)', 0);
SELECT @g.STIntersects(@h);--是否相交
--由于两条线相交，所以返回：1
```

4. STTouches()方法确定两个实例是否接触

STTouches（other_geometry）方法表示如果一个 geometry 实例在空间上与另一个 geometry 实例接触，则返回 1；否则，返回 0。与相交的判断函数 STIntersects()不同的是，

如果两个 geometry 实例的点集相交，但是它们的内部不相交，则表明这两个实例接触。可以认为接触是一种特殊的相交。例如两个 LineString 实例，判断其是否是接触的脚本如代码 15.50 所示。

代码 15.50　判断 2 个实例是否接触

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 2, 2 0)', 0);
SET @h = geometry::STGeomFromText('LINESTRING(2 2,1 1)', 0);
SELECT @g.STTouches(@h); --两实例相接触，返回 1
SET @h = geometry::STGeomFromText('LINESTRING(2 2,0 0)', 0);
SELECT @g.STTouches(@h); --两实例未接触，返回 0
```

注意：对于 LineString 对象，只有其端点与其他对象相交，而内部不相交才被判断为接触。

5．STOverlaps()方法确定两个实例是否重叠

STOverlaps（other_geometry）方法表示如果 geometry 实例与另一个 geometry 实例重叠，则返回 1；否则，返回 0。如果表示这两个 geometry 实例交集的区域与这两个实例具有相同的维度，而且这两个实例不相等，则说明这两个实例重叠。如果两个 geometry 实例的交点与这两个实例具有不同的维度，则 STOverlaps()方法始终返回 0。例如两个 Polygon 实例，判断其是否重叠的脚本如代码 15.51 所示。

代码 15.51　判断 2 个实例是否重叠

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 2 0, 2 2, 0 2, 0 0))', 0);
SET @h = geometry::STGeomFromText('POLYGON((1 1, 3 1, 3 3, 1 3, 1 1))', 0);
SELECT @g.STOverlaps(@h);        --是否重叠
--系统返回结果：1
```

6．STCrosses()方法确定两个实例是否交叉

STCrosses（other_geometry）方法表示如果 geometry 实例与另一个 geometry 实例相交，则返回 1；否则，返回 0。

如果下面的这两个条件均为真，则这两个 geometry 实例将交叉。

- 这两个 geometry 实例的交集会生成一个维度小于源 geometry 实例最大维度的几何图形。
- 交集位于这两个源 geometry 实例的内部。

例如两个 LineString 实例，判断是否交叉的脚本如代码 15.52 所示。

代码 15.52　判断 2 个实例是否交叉

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 2, 2 0)', 0);
SET @h = geometry::STGeomFromText('LINESTRING(2 2,1 1)', 0);
SELECT @g.STCrosses(@h); --两实例相交点是其中一个实例的边界，返回 0
```

```
SET @h = geometry::STGeomFromText('LINESTRING(2 2,0 0)', 0);
SELECT @g.STCrosses(@h); --两实例相交叉，返回 1
```

7．STWithin()方法确定某个实例是否在另一个实例内部

STWithin（other_geometry）方法表示如果 geometry 实例完全包含在另一个 geometry 实例中，则返回 1；否则返回 0。所谓内部是指一个实例的所有点集都在另一个实例内部。

注意：一个实例与另一个实例如果有共同的边界，则不能算是在另一个实例的内部。

例如两个 Polygon 实例，一个实例是另外一个实例的左半边部分，则判断一个实例是否在另一个实例内部的脚本如代码 15.53 所示。

代码 15.53　判断一个实例是否在另一个实例内部

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 2 0, 2 2, 0 2, 0 0))', 0);
SET @h = geometry::STGeomFromText('POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))', 0);
SELECT @g.STWithin(@h);--是否在实例内部
--由于有共同的部分边界，所以系统返回 0
```

8．STContains()方法确定某个实例是否包含另一个实例

STContains（other_geometry）方法表示如果 geometry 实例完全包含另一个 geometry 实例，则返回 1；否则返回 0。与 STWithin()方法不同，STContains()方法中如果一个实例的所有点集都属于另一个实例的所有点集包括边界，则认为是包含关系。同样是两个 Polygon 实例，一个实例是另外一个实例的左半边部分，则判断一个实例是否包含另一个实例的脚本如代码 15.54 所示。

代码 15.54　判断一个实例是否包含另一个实例

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 2 0, 2 2, 0 2, 0 0))', 0);
SET @h = geometry::STGeomFromText('POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))', 0);
SELECT @g.STContains(@h);       --是否包含了另一个实例
--系统返回：1
```

9．STRelate()方法确定两个实例是否存在空间关系

STRelate（other_geometry,intersection_pattern_matrix）方法表示如果一个 geometry 实例与另一个 geometry 实例相关（其关系是由“维扩展 9 交集模型”（DE-9IM）模式矩阵值定义的），则返回 1；否则返回 0。第二个参数 intersection_pattern_matrix 表示两个 geometry 实例间 DE-9IM 模式矩阵设备的 nchar(9)类型编码可接受字符串。

DE-9IM 是 The Dimensionally Extended Nine-Intersection Model 的缩写，要使用 DE-9IM，首先要建立几何对象的 interior、boundary 和 exterior，即对象的内部、边界和外部。举例来说一个点的 boundary 为空，未封闭线的 boundary 为其两个端点，封闭线的 boundary 为空，多边形的 boundary 为它的环状边界。

约定：以 I(a)，B(a)，E(a)表示几何对象 a 的 interior、boundary 和 exterior，以 dim(a)

表示几何对象的维度，在二维空间中它的取值为{–1, 0, 1, 2}，其中–1 代表空。为了便于表示，用下面一些符号来表示取值的集合：

- T : {0, 1, 2}；
- F : {–1}；
- * : {–1, 0, 1, 2}；
- 0 : {0}；
- 1 : {1}；
- 2 : {2}。

那么就可以用这样一个矩阵来判断几何对象 a 和 b 的位置关系，如表 15.1 所示。

表 15.1　几何对象a和b的位置关系

	Interior	Boundary	Exterior
Interior	dim(I(a)∩I(b))	dim(I(a)∩B(b))	dim(I(a)∩E(b))
Boundary	dim(B(a)∩I(b))	dim(B(a)∩B(b))	dim(B(a)∩E(b))
Exterior	dim(E(a)∩I(b))	dim(E(a)∩B(b))	dim(E(a)∩E(b))

一般将零维对象简写为 P，一维对象简写为 L，二维对象简写为 A。前面介绍的对象之间是否相交、是否包含、是否接触等关系，都可以通过 DE-9IM 来表示。

- Equal：两个几何对象完全相同，它的定义表示为 DE-9IM 即"TFFFTFFFT"。
- Disjoint：两个对象的边界和内部都没有任何公共部分，表示为 DE-9IM 即"FF*FF****"。
- Intersection Disjoint：取反。
- Touches：简单地说，Touches 表示两个对象的边缘相接触，这个关系是 A/A、L/L、L/A、P/A、P/L。用 DE-9IM 表示，可以为"FT*******"、"F**T*****"或"F***T****"几类几何对象间特有的。
- Crosses Crosses：表示一个对象穿过另一个对象，它应用于 P/L、P/A、L/L 和 L/A 之间。用 DE-9IM 表示为"T*T******"（P/L、P/A、L/A），"0********"（L/L）。
- Within：包含于。用 DE-9IM 表示为"T*F**F***"。
- Overlaps：相叠，应用于 A/A、L/L 和 P/P 之间。用 DE-9IM 表示为"T*T***T**"（A/A，P/P）、"1*T***T**"（L/L）。
- Contains：包含，对立于 Within。

例如，一个 LineString 对象和一个 Point 对象，使用 STRelate 判断两个对象边界和内部都没有任何公共部分的脚本，如代码 15.55 所示。

代码 15.55　使用 DE-9IM 判断两个实例关系

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('LINESTRING(0 2, 2 0, 4 2)', 0);
SET @h = geometry::STGeomFromText('POINT(5 5)', 0);
SELECT @g.STRelate(@h, 'FF*FF****');    --实例关系判断
--系统返回结果：1
```

10．STDistance()方法确定两个几何图形中的点之间的最短距离

STDistance（other_geometry）方法表示返回一个 geometry 实例中的点，与另一个

geometry 实例中的点之间的最短距离。例如获得一个点到一条线的最短距离的脚本如代码 15.56 所示。

代码 15.56　获得实例之间最短距离

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('LineString(0 0, 2 0, 2 2, 0 2)', 0);
SET @h = geometry::STGeomFromText('POINT(10 10)', 0);
SELECT @g.STDistance(@h);      --实例之间最短距离
--系统返回结果：11.3137084989848
```

15.3　geography 地理数据类型

地理空间数据类型 geography 表示圆形地球坐标系中的数据，是作为 SQL Server 中的.NET 公共语言运行时（CLR）数据类型实现的。SQL Server geography 数据类型用于存储诸如 GPS 纬度和经度坐标之类的椭球体（圆形地球）数据。

15.3.1　创建地域实例

地域实例的构造方法与几何实例的构造方法相同，只是地域构造实例使用 geography 下的方法，而几何实例的构造使用的是 geometry 下的方法。

1. 构造

geography 数据类型提供了如下若干种用开放地理空间联盟（OGC）WKT 表示形式生成地域的内置方法。

- 用 WKT 输入构造任意类型的地域实例 STGeomFromText()(geography 数据类型)；
- 用 WKT 输入构造地域 Point 实例 STPointFromText()（geography 数据类型）；
- 用 WKT 输入构造地域 MultiPoint 实例 STMPointFromText()(geography 数据类型)；
- 用 WKT 输入构造地域 LineString 实例 STLineFromText()（geography 数据类型）；
- 用 WKT 输入构造地域 MultiLineString 实例 STMLineFromText()（geography 数据类型）；
- 用 WKT 输入构造地域 Polygon 实例 STPolyFromText()（geography 数据类型）；
- 用 WKT 输入构造地域 MultiPolygon 实例 STMPolyFromText()（geography 数据类型）；
- 用 WKT 输入构造地域 GeometryCollection 实例 STGeomCollFromText()(geography 数据类型）。

例如，通过 WKT 构造一个 LineString 的 geography 实例，如代码 15.57 所示。

代码 15.57　通过 WKT 构造 geography 实例

```
DECLARE @g geography;
SET @g = geography::STGeomFromText('LINESTRING (-122.36 47.656, -122.343
47.656)', 4120);
--定义地理空间数据
```

注意：地域类型的 WKT 中与几何类型的 WKT 有所不同，地域类型 WKT 中第一个数值是经度，经度值必须在–15 069～15 069 度之间。第二个数值是纬度，纬度值必须在–90～90 度之间。

除了使用 WKT 构造地域实例外，还可以通过 WKB 构造 geography 数据实例。以下函数接受使用 WKB 输入构造地域实例。

- 用 WKB 输入构造任意类型的地域实例 STGeomFromWKB()（geography 数据类型）；
- 用 WKB 输入构造地域 Point 实例 STPointFromWKB()（geography 数据类型）；
- 用 WKB 输入构造地域 MultiPoint 实例 STMPointFromWKB()（geography 数据类型）；
- 用 WKB 输入构造地域 LineString 实例 STLineFromWKB()（geography 数据类型）；
- 用 WKB 输入构造地域 MultiLineString 实例 STMLineFromWKB()（geography 数据类型）；
- 用 WKB 输入构造地域 Polygon 实例 STPolyFromWKB()（geography 数据类型）；
- 用 WKB 输入构造地域 MultiPolygon 实例 STMPolyFromWKB()（geography 数据类型）；
- 用 WKB 输入构造地域 GeometryCollection 实例 STGeomCollFromWKB()（geography 数据类型）。

例如，通过 WKB 构造一个地域实例如代码 15.58 所示。

代码 15.58　通过 WKB 构造 geography 实例

```
DECLARE @g geography;
SET @g =geography::STGeomFromWKB(
0x010300000001000000050000000000000000000000000000000000000000000000000
00040000000000000000000000000000004000000000000000400000000000000000000
0000000000004000000000000000000000000000000000, 4326);
```

除了使用 WKT 和 WKB 构造地域实例外，还可以通过 GML 构造地域实例。通过 GML 构造地域实例使用 GeomFromGML()函数，其格式为：

```
GeomFromGml ( GML_input, SRID )
```

其中 GML_input 是 XML 输入，GML 将从该输入返回值。SRID 为一个 int 表达式，它表示用户希望返回的 geography 实例的空间引用 ID（SRID）。例如通过 GML 构造一个 LineString 的实例，如代码 15.59 所示。

代码 15.59　通过 GML 构造地域实例

```
DECLARE @g geography;
DECLARE @x xml;
SET @x = '<LineString xmlns="http://www.opengis.net/gml"><posList>47.656
-122.36 47.656 -122.343</posList></LineString>';   --定义 GML
SET @g = geography::GeomFromGml(@x, 4120);          --通过 GML 构造实例
SELECT @g.ToString();   --获得地理空间数据的 WKT
--系统输出结果：
LINESTRING (-122.36 47.656, -122.343 47.656)
```

2. 输出

对于 geography 实例，可以通过其实例方法将实例的 WKT、WKB 和 GML 输出。系

统提供的方法有：

- ❑ 返回一个 geography 实例的 WKT 表示形式 STAsText()（geography 数据类型）和 ToString()（geography 数据类型）；
- ❑ 返回一个 geography 实例的 WKT 表示形式，同时还包含 Z 和 M 值 AsTextZM()（geography 数据类型）；
- ❑ 返回地域实例的 WKB 表示形式 STAsBinary()（geography 数据类型）；
- ❑ 返回地域实例的 GML 表示形式 AsGml()（geography 数据类型）。

例如构建一个地域实例，获得其 WKT、WKB 和 GML 的脚本如代码 15.60 所示。

代码 15.60　获得地域实例的 WKT、WKB 和 GML

```
DECLARE @g geography;
SET @g = geography::STGeomFromText('LINESTRING (-122.36 47.656, -122.343
47.656)', 4120);
SELECT @g.ToString();   --输出 WKT
SELECT @g.STAsBinary();--输出 WKB
SELECT @g.AsGml();      --输出 GML
--系统返回结果：
LINESTRING (-122.36 47.656, -122.343 47.656)
0x01020000000200000D7A3703D0A975EC08716D9CEF7D34740CBA145B6F3955EC0871
6D9CEF7D34740
<LineString xmlns="http://www.opengis.net/gml"><posList>47.656 -122.36
47.656 -122.343</posList></LineString>
```

3. 查询实例类型和GeometryCollection信息

在构造了 geography 实例后，可以通过下列方法返回实例类型或者返回特定的实例。

- ❑ 返回地域的实例类型 STGeometryType()（geography 数据类型）；
- ❑ 确定地域是否为给定的实例类型 InstanceOf()（geometry 数据类型）；
- ❑ 返回组成 geography 实例的 geometries 的个数 STNumGeometries()（geography 数据类型）；
- ❑ 返回 GeometryCollection 实例中的特定地域 STGeometryN()（geography 数据类型）。

这几个方法在介绍 geometry 数据类型时已经举例讲解，这里不再重复讲解。

15.3.2　地域实例的属性和方法

所有 geography 实例都有很多可以通过 SQL Server 提供的方法进行检索的属性。下列为主要介绍地域类型的属性和方法。

1. 点数

所有非空 geography 实例都由“点”组成。这些点表示球体的纬度和经度坐标，在其上可绘制 geography 实例。数据类型 geography 提供了许多用于查询实例点的内置方法。

- ❑ 返回构成实例的点数 STNumPoints()（geography 数据类型）；

- ❑ 返回实例中的特定点 STPointN()（geometry 数据类型）；
- ❑ 返回实例的起始点 STStartPoint()（geography 数据类型）；
- ❑ 返回实例的终点 STEndpoint()（geography 数据类型）。

这几个方法在前面的 geometry 数据类型中已经举例介绍，其使用方法与 geometry 数据类型调用方法相同。这里举个例子说明其使用。例如要获得一个 LineString 地域数据类型的起点，则对应的脚本如代码 15.61 所示。

代码 15.61　获得 LineString 地域数据类型的起点

```
DECLARE @g geography;
SET @g = geography::STGeomFromText('LINESTRING (1 1,1 2)', 4326);
SELECT @g.STStartPoint().ToString();   --获得起点的 WKT
--系统返回结果：POINT (1 1)
```

2．维度

非空 geography 实例可以为零维、一维或二维。零维 geography 实例（例如 Point 和 MultiPoint）没有长度或面积。一维对象（例如 LineString 和 MultiLineString）具有长度。二维实例（例如 Polygon 和 MultiPolygon）具有面积和长度。空实例将报告–1 维，并且 GeometryCollection 报告其内容的最大维度。同 geometry 类型相同，系统提供了以下方法：

- ❑ 返回实例的维度 STDimension()（geography 数据类型）；
- ❑ 返回实例的长度 STLength()（geography 数据类型）；
- ❑ 返回实例的面积 STArea()（geography 数据类型）。

这里需要注意的是，在 geometry 数据类型中使用 XY 坐标表示，其长度、面积都是以单位长度为单位。而在 geography 数据类型中，使用经度纬度来表示，其长度、面积默认都是以米为单位。例如要求两个地域数据类型点之间的距离，其脚本如代码 15.62 所示。

代码 15.62　获得 LineString 地域类型的长度

```
DECLARE @g geography;
SET @g = geography::STGeomFromText('LINESTRING(0 0,0 1)', 4326);
SELECT @g.STLength(); --系统返回：110574.388493406
SET @g = geography::STGeomFromText('LINESTRING(0 0,1 0)', 4326);
SELECT @g.STLength(); --系统返回：111319.490735885
```

注意：在地域数据类型中使用经度和纬度表示，所以 0 0 到 0 1 的距离与 0 0 到 1 0 的距离是不相同的。

3．空

“空”geography 实例不包含任何点。空 LineString 和 MultiLineString 实例的长度为 0。空 Polygon 和 MultiPolygon 实例的面积为 0。SQL Server 提供 STIsEmpty()方法用于判断一个 geography 实例是否为空。例如创建一个 geography 对象实例，判断其是否为空的脚本如代码 15.63 所示。

代码 15.63　判断 geography 实例是否为空

```
DECLARE @g geography;
SET @g = geography::STGeomFromText('POLYGON EMPTY', 4326);
```

```
SELECT @g.STIsEmpty();  --判断地理数据是否为空
--系统返回结果：1
```

4．闭合

“闭合的”geography 实例是指起始点和终点相同的图形。Polygon 实例是闭合的。Point 实例不是闭合的。环是一个简单、闭合的 LineString 实例。判断一个对象是否关闭得使用 STIsClosed()函数。例如判断一个 geography 实例是否为闭合的脚本如代码 15.64 所示。

代码 15.64　判断 geography 实例是否闭合

```
DECLARE @g geography;
SET @g = geography::STGeomFromText('POLYGON((0 0,1 0,1 1,0 1,0 0))', 4326);
SELECT @g.STIsClosed();  --判断地理数据是否闭合
--系统返回结果：1
```

NumRings()返回 Polygon 实例中的总环数。在 SQL Server geography 类型中，由于可以将任何环视为外部环，因此不对外部环和内部环进行区分。

如果该实例不是 Polygon 实例，则此方法返回 Null；如果该实例为空，则将返回 0。此方法是精确方法。例如构建一个 Polygon 实例，获得总环数的脚本如代码 15.65 所示。

代码 15.65　获得 geography 实例的总环数

```
DECLARE @g geography;
SET @g = geography::STGeomFromText('POLYGON((0 0,2 0,2 2,0 2,0 0),(1 1,1
1.5,1.5 1.5,1.5 1,1 1))', 4326);
SELECT @g.NumRings();       --获得地理数据类型总环数
--系统返回结果：2
```

RingN(expression)返回 geography 实例的指定环。其参数 expression 是一个 int 表达式，其值介于 1 和 polygon 实例中的环数之间。

例如，获得一个 Polygon 实例的第 2 个环的脚本，如代码 15.66 所示。

代码 15.66　获得 geography 实例的某个环

```
DECLARE @g geography;
SET @g = geography::STGeomFromText('POLYGON((0 0,2 0,2 2,0 2,0 0),(1 1,1
1.5,1.5 1.5,1.5 1,1 1))', 4326);
SELECT @g.RingN(2).ToString();  --获得某个环
--系统返回结果：
LINESTRING (1 1, 1 1.5, 1.5 1.5, 1.5 1, 1 1)
```

5．空间引用标识符

地域类型实例的属性 STSrid 是一个表示该实例的空间引用标识符（SRID）的整数。前面已经讲到，对于地域数据类型 SRID 必须为 sys.spatial_reference_systems 视图中的一个值，默认情况下 SRID 为 4326。通过属性 STSrid，获得和修改一个 geography 数据类型的 SRID 脚本如代码 15.67 所示。

代码 15.67　获得和修改 geography 数据类型的 SRID

```
DECLARE @g geography;
SET @g = geography::STGeomFromText('POLYGON((0 0,2 0,2 2,0 2,0 0),(1 1,1
1.5,1.5 1.5,1.5 1,1 1))', 4326);
```

```
SELECT @g.STSrid;  --4326
SET @g.STSrid = 4267;
SELECT @g.STSrid;  --4327
```

15.3.3　地域实例之间的关系

geography 数据类型提供了许多内置方法，可以使用这些方法确定两个 geography 实例的关系。

- 确定两个实例是否包含相同的点集 STEquals()（geometry 数据类型）；
- 确定两个实例是否不相接 STDisjoint()（geometry 数据类型）；
- 确定两个实例是否相交 STIntersects()（geometry 数据类型）；
- 确定两个实例的交点 STIntersection()（geography 数据类型）；
- 确定两个地域实例中点之间的最短距离 STDistance()（geometry 数据类型）；
- 确定两个地域实例之间点的不同 STDifference()（geography 数据类型）；
- 派生一个地域实例相比于另一个地域实例的余集或唯一点 STSymDifference()（geography 数据类型）。

这几个方法在前面介绍 geometry 数据类型时已经举例介绍，在 geography 数据类型中的使用方法与 geometry 实例中的使用方法相同。例如获得一个 Point 实例到一个 Polygon 实例的最短距离，如代码 15.68 所示。

代码 15.68　获得 geography 数据类型 Point 到 Polygon 的距离

```
DECLARE @g geometry;
DECLARE @h geometry;
SET @g = geometry::STGeomFromText('POLYGON((0 0, 2 0, 2 2, 0 2, 0 0))', 0);
SET @h = geometry::STGeomFromText('POINT(10 10)', 0);
SELECT @g.STDistance(@h);        --返回地理空间数据之间的距离
--系统返回的结果为：11.3137084989848
```

15.4　空　间　索　引

“空间索引”是一种扩展索引，允许对空间列编制索引。空间列是包含空间数据类型（如 geometry 或 geography）数据的表列。本节中主要介绍空间索引。

15.4.1　空间索引概述

1．将索引空间分解成网格层次结构

在 SQL Server 2012 中，空间索引与一般数据的索引一样，都是使用 B 树构建而成，也就是说，这些索引必须按 B 树的线性顺序表示二维空间数据。在将数据读入空间索引之前，SQL Server 2012 先实现对空间的分层均匀分解。空间索引创建过程会将空间分解成一个四级“网格层次结构”。这些级别指的是“第 1 级”（顶级）、“第 2 级”、“第 3 级”和“第 4 级”。

每个后续级别都会进一步分解其上一级，这样按层次级别进行逐步分解，级别越高分

成的网格就越多，上一级别的每个单元都包含下一级别的整个网格。在同一个级别上，所有网格沿两个轴都有相同数目的单元（例如 4×4 或 8×8），不会出现 4×8 之类的分割方式，并且每个网格单元的大小都相同。

如图 15.11 显示了网格层次结构每个级别的右上角单元被分解成 4×4 网格的情况。对于每个网格单元都是以这种方式进行逐步分解的。以此为例，将一个空间分解成四个级别的 4×4 网格，实际上总共会产生 65 536 个第四级单元。

空间在被分成网格后，网格层次结构的单元就利用多种 Hilbert 空间填充曲线以线性方式编号。例如在图 15.12 中，几个表示建筑物的多边形和表示街道的线所在的二维空间已经分割为一个 4×4 的 1 级网格。第 1 级单元的编号为 1～16，编号从左上角的单元开始。

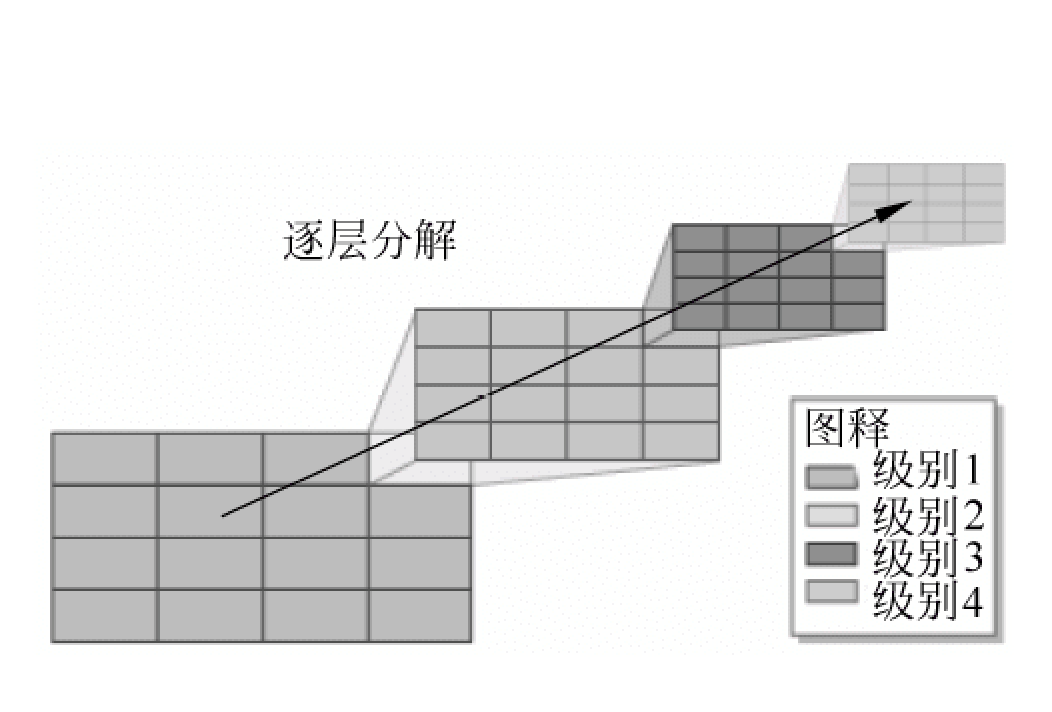


图 15.11　网格层次结构

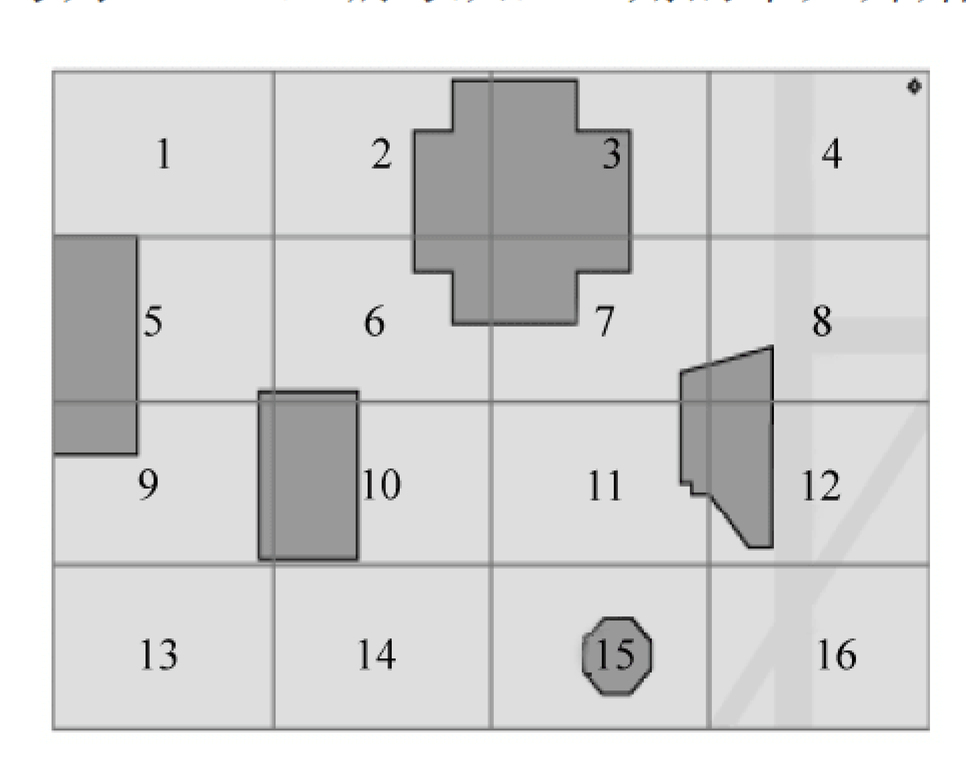


图 15.12　网格编号

一个二维空间可以被分割成 4×4、8×8 或者 16×16，不同的分割方案确定了网格的“密度”：单元数目越大，网格的密度越大。例如，8×8 网格（产生 64 个单元）的密度就大于 4×4 网格（产生 16 个单元）的密度。在网格的 4 个层次结构中可以为每个层次定义网格密度。

CREATE SPATIAL INDEX 语句支持 GRIDS 子句，使用该子句可以在不同级别指定不同的网格密度。如表 15.2 显示了关键字指定给定级别的网格密度。默认使用 8×8 的分割方案，设置所有级别都为 MEDIUM。

表 15.2　网格密度

关键字	网格配置	单元数目	关键字	网格配置	单元数目
LOW	4×4	16	HIGH	16×16	256
MEDIUM	8×8	64			

2．分割

将索引空间分解成网格层次结构后，空间索引将逐行读取空间列中的数据。读取空间对象（或实例）的数据后，空间索引将为该对象执行“分割过程”。分割过程通过将对象与其接触的网格单元集（“接触单元”）相关联使该对象适合网格层次结构。分割过程也是逐级进行的，从网格层次结构的第 1 级开始，分割过程以“广度优先”方式对整个级别进行处理。在可能的情况下，分割过程可以连续处理所有四个级别，一次处理一个级别。

分割过程将输出对象的空间索引中所记录的接触单元集。通过引用这些已记录单元，空间索引可以确定该对象在空间中相对于空间列中也存储在索引中的其他对象的位置。

为了限制为对象记录的接触单元数，分割过程采用了几个分割规则。这些规则确定分割过程的深度及在索引中记录哪些接触单元。这些规则如下：

（1）覆盖规则。如果一个对象完全盖住了某个单元，则称该单元由该对象所“覆盖”。被覆盖的单元会参与计数，但不进行分割。此规则应用于网格层次结构的所有级别。覆盖规则简化了分割过程，并减少了空间索引记录的数据量。如图 15.13 所示，图中一个第 2 级单元 15.11 完全由八边形的中间部分所覆盖，那么在 3 级的分割过程中将不会再对单元格 15.11 进行分割，而是参与计数并记录在索引中。

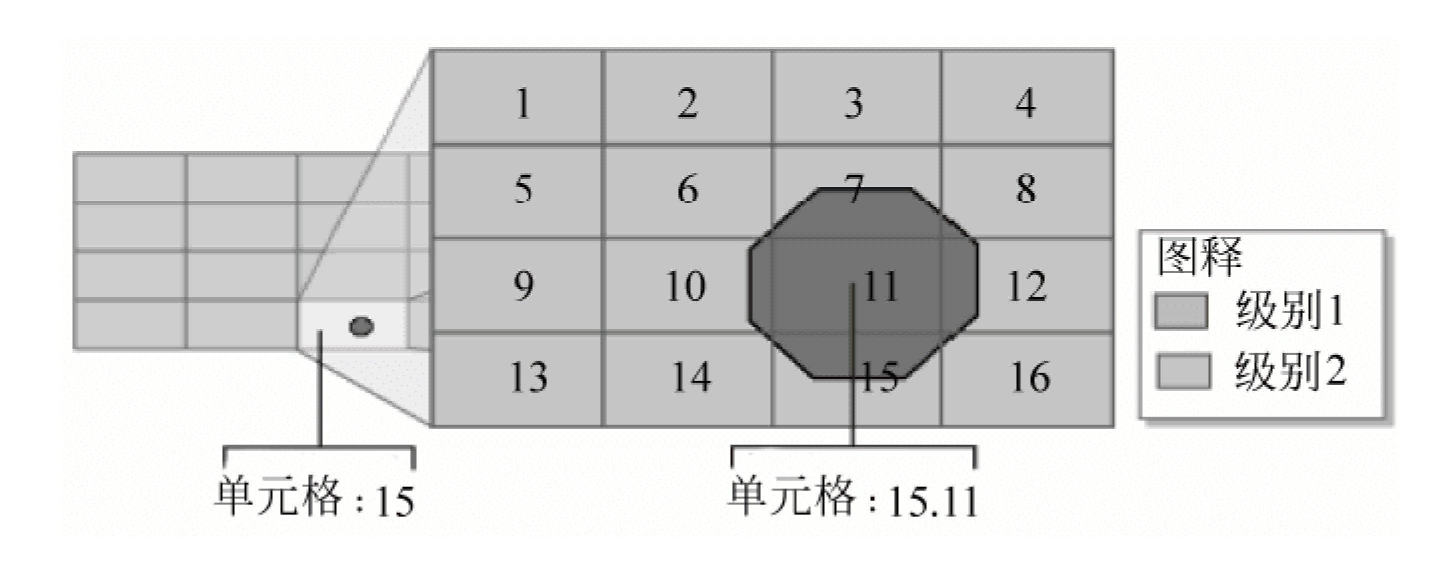


图 15.13　覆盖规则

（2）每对象单元数规则。此规则强制执行“每对象单元数限制”，这将确定可以为每个对象计数的最大单元数（第 1 级除外）。在较低级别，每对象单元数规则会控制可以记录的有关对象的信息量。每个对象的分割程度主要取决于空间索引的“每对象单元数限制”。此限制确定了对于每个对象分割可以计数的最大单元数。然而，每对象单元数规则不对第 1 级强制执行，因此该级可能超出此限制。如果第 1 级计数达到（或超出）每对象单元数限制，则在较低级别不再进行分割。

只要计数低于每对象单元数限制，分割过程就将继续。例如图 15.13，假设现在单元数限制为 10，级别 2 中接触单元数为 9，所以需要继续分割，分割从单元 15.6 编号最低的接触单元开始，此过程将测试每个单元以评估是对其进行计数还是进行分割。单元 15.11 如果在级别 3 上进行分割，则接触单元数为 16，大于了限制 10，所以直接为单元 15.11 计数，不再进行分割，这也与前面讲到的覆盖原则得出的结果相同。对于其他的单元格，将对该单元进行分割，而对由对象接触的较低级别的单元进行计数。分割过程将以这种方式在整个级别的广度范围内继续进行。此过程对低级别网格的分割单元依次逐步进行重复，直至达到限制或不再有要计数的单元为止。

默认情况下，每对象单元数限制为每个对象 16 个单元，这将在大多数空间索引的空间和精度之间提供一个令人满意的折中方案。然而，CREATE SPATIAL INDEX 语句支持 CELLS_PER_OBJECT=n 子句，使用该子句可以指定介于 1～8192（包含这两者）之间的每对象单元数限制。

注意： 空间索引的 cells_per_object 设置，显示在 sys.spatial_index_tessellations 目录视图中。

（3）最深单元规则。最深单元规则通过只记录已为对象分割的最底部单元生成该对象的最近似对象。父单元不计入每对象单元数，这些单元不记录在索引中。这些分割规则依次逐步应用于每个网格级别。

最深单元规则利用每个较低级别单元属于其上级单元这一事实：第 4 级单元属于第 3 级单元，第 3 级单元属于第 2 级单元，第 2 级单元属于第 1 级单元。例如，属于单元 1.1.1.1 的对象也属于单元 1.1.1、单元 1.1 以及单元 1。这种单元层次结构关系知识已内置于查询处理器中。因此，只有最深级别的单元需要记录在索引中，从而最大限度地减少了索引需要存储的信息。

例如在图 15.14 中，要对一个相对较小的多边形进行分割建立索引。索引使用默认的每对象单元数限制 16，此对象较小，在级别 1～3 中都未达到该限制。因此，分割一直向下继续到第 4 级。此多边形驻留在以下的第 1 级到第 3 级的单元中：4、4.4 及 4.4.10 和 4.4.14。根据最深单元规则，分割将仅对 12 个位于第 4 级的单元进行计数：4.4.10.13-15 及 4.4.14.1-3、4.4.14.5-7 和 4.4.14.9-11，其他单元格不计数。

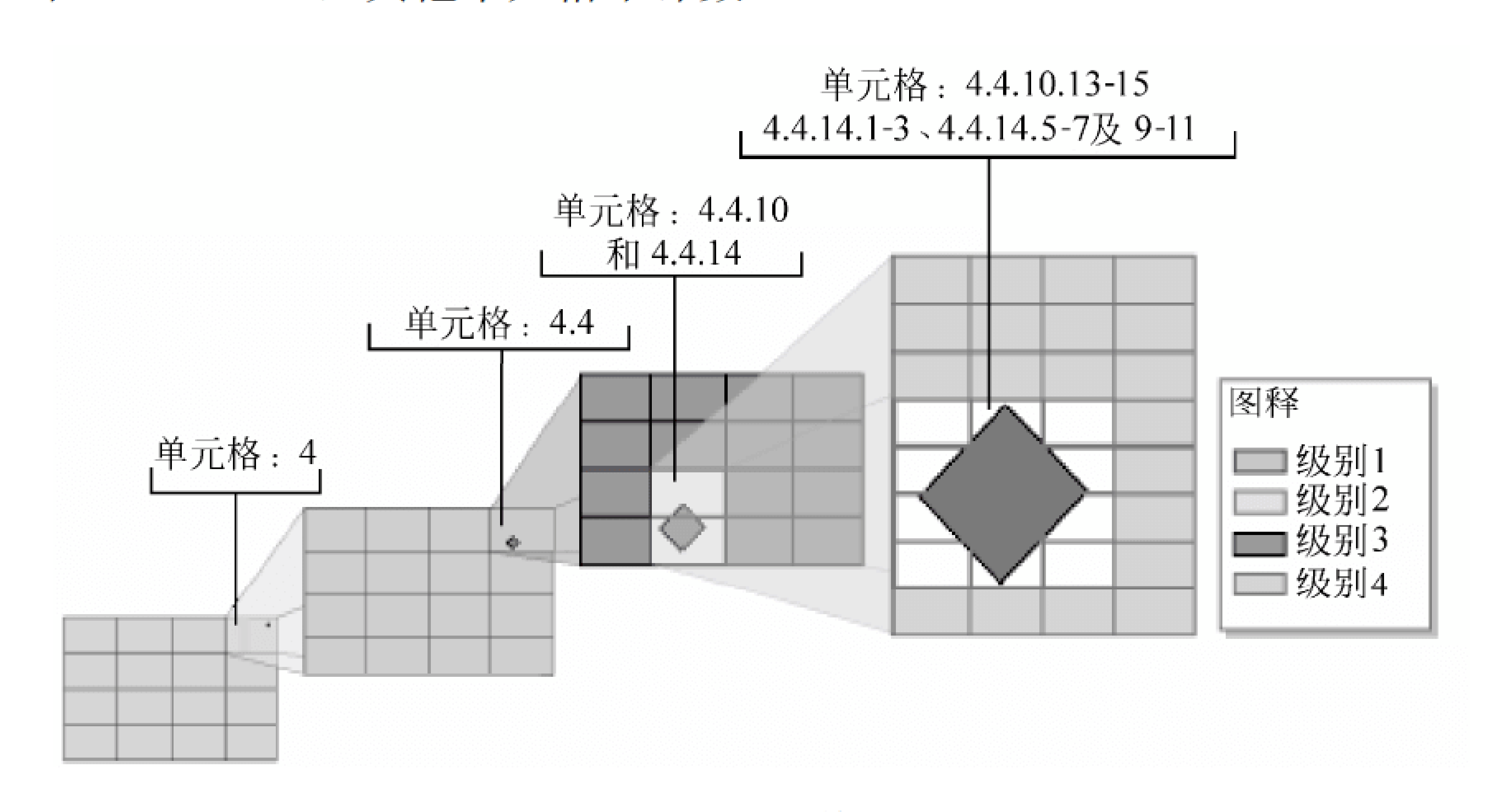


图 15.14　最深单元规则

3．几何图形分割方案

空间索引的行为部分取决于“分割方案”。分割方案特定于数据类型。在 SQL Server 2012 中，分为几何数据类型和地理数据类型，对应的空间索引也支持如下两种分割方案。

- ❑ 几何图形网格分割，这是适用于 geometry 数据类型的方案。
- ❑ 地理网格分割方案，该方案适用于数据类型为 geography 的列。

使用空间索引是通过充当对象筛选器，来减少将面向集合的方法应用于空间列的开销。geometry 数据类型提供了一些内置的方法，以构造用于描述几何对象的 geometry 实例，并使用这些实例。在特定条件下，空间索引支持以下面向集合的几何图形方法，分别是 STContains()、STDistance()、STEquals()、STIntersects()、STOverlaps()、STTouches()和 STWithin()。若要使空间索引支持这些方法，必须在查询的 WHERE 子句中使用这些方法。

几何数据占有的平面可以是无限的。然而，在 SQL Server 2012 中，空间索引需要有限空间。几何图形网格分割方案中使用矩形“边界框”来建立有限空间以用于分解。该边界框由 4 个坐标（x-min, y-min）和（x-max, y-max）定义，这些坐标存储为空间索引的属性。（x-min, y-min）和（x-max, y-max）坐标确定边界框的位置和尺寸。边界框的外部空间视作一个编号为 0 的单元。

空间索引以边界框中的矩形为基础，对其进行分解。网格层次结构的第 1 级网格将填充边界框。若要在网格层次结构中放置几何对象，空间索引会将该对象的坐标与边界框的

坐标进行比较。只有针对完全位于边界框内部对象的计算操作才会受益于空间索引。因此，若要获得 geometry 列的空间索引所能提供的最大优势，就需要指定一个包含所有或大多数对象的边界框。

4．地理网格分割方案

地理网格分割方案仅适用于 geography 列。此部分总结了地理网格分割支持的方法，并讨论了如何将测量空间投影到平面上，该平面随后将分解成网格层次结构。在某些条件下，空间索引支持以下面向集合的地域方法，分别是 STIntersects()、STEquals()和 STDistance()。若要使空间索引支持这些方法，必须在查询的 WHERE 子句中使用这些方法。

由于 geography 实例（对象）的空间被视作椭圆体，而分解空间是在平面上进行，所以若要分解此空间，地理网格分割方案将椭圆体表面分为上半球和下半球，然后执行下列步骤：

（1）将每个半球投影在四边形棱锥图面上。

（2）将两个棱锥图平展开。

（3）连接平展的棱锥图以形成非欧几里得平面。

如图 15.15 显示了此三步分解过程的示意图。在棱锥图中，虚线表示每个棱锥图的 4 个面的边界。步骤（1）和步骤（2）显示测量椭圆体，使用一条绿色水平线表示赤道经线，使用一系列绿色垂直线表示若干条纬线。步骤（1）显示要投影在两个半球上的棱锥图。步骤（2）显示要平展的棱锥图。步骤（3）显示平展的棱锥图，这些棱锥图已组合起来形成一个平面，显示出许多投影的经线。请注意，这些投影线伸直后长度不一，具体取决于它们落在棱锥图上的位置。

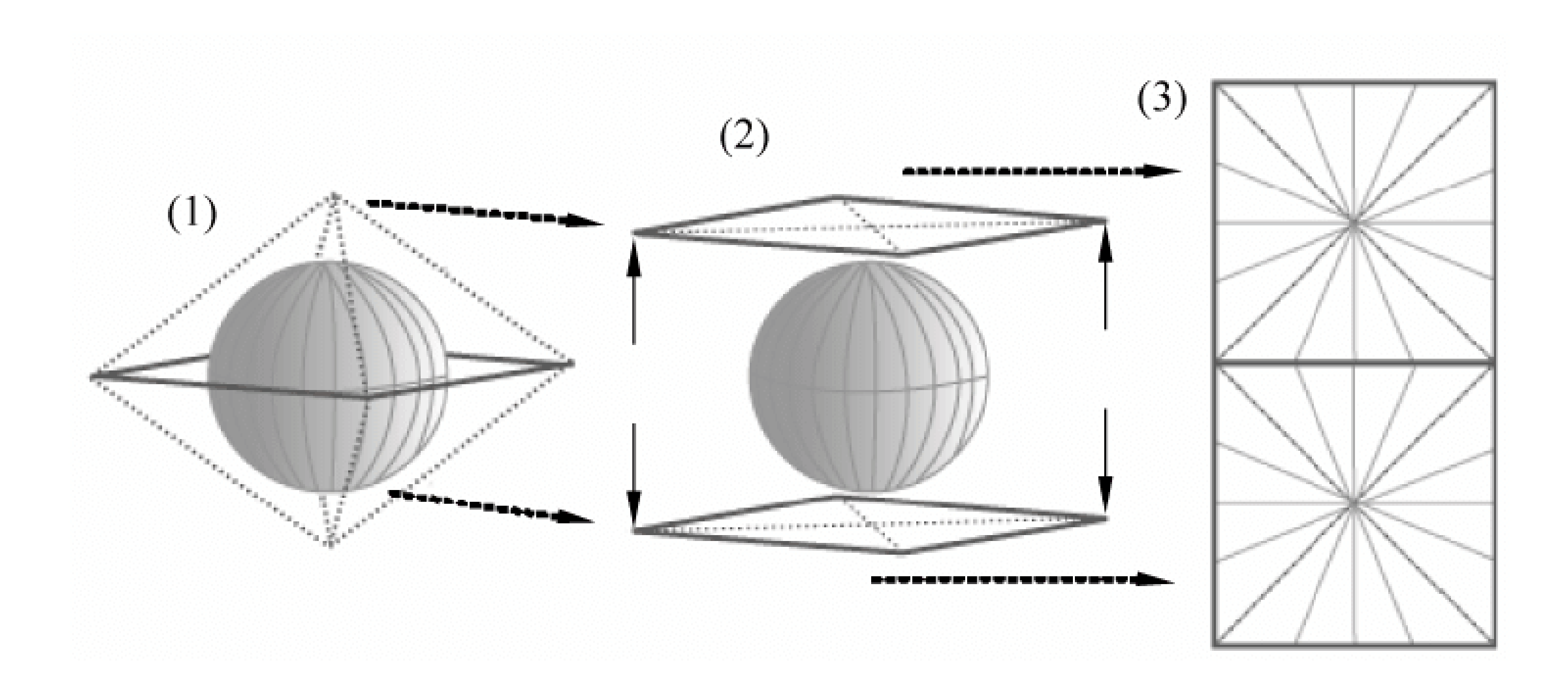


图 15.15　地理网格分割

空间投影到平面上之后，此平面将会分解成四级网格层次结构。不同级别可以使用不同的网格密度。此平面完全分解成了一个四级网格层次结构。分解过程完成后，将逐行从 geography 列读取地理数据，并为每个对象依次执行分割过程。

5．空间索引限制

在使用空间索引时存在以下限制：

- 只能对类型为 geometry 或 geography 的列创建空间索引。
- 只能对具有主键的表定义空间索引。表中主键列的最大数目为 15。
- 索引键记录的最大大小为 895 字节。超过此大小会引发错误。

- ❑ 对表定义空间索引时，不能更改主键元数据。
- ❑ 不能对索引视图指定空间索引。
- ❑ 最多可对支持的表中的任何空间列创建 249 个空间索引。对同一空间列创建多个空间索引可能很有用，例如，在要对单个列中的不同分割参数进行索引时。
- ❑ 一次只能创建一个空间索引。

15.4.2　使用 T-SQL 创建空间索引

空间索引只能对类型为 geometry 或 geography 的列创建空间索引。所以在创建空间索引的示例之前，需要创建一个含有空间数据类型的列的表。创建该表的脚本如代码 15.69 所示。

代码 15.69　创建空间数据类型表

```
CREATE TABLE GeometryTest   --创建测试表
(
    ID int IDENTITY PRIMARY KEY,
    GeometryValue geometry NOT NULL
)
CREATE TABLE GeographyTest
(
    ID int IDENTITY PRIMARY KEY,
    GeographyValue geography NOT NULL
)
```

SQL Server 为创建空间索引提供了 CREATE SPATIAL INDEX 命令，该命令的语法格式如代码 15.70 所示。

代码 15.70　CREATE SPATIAL INDEX 语法

```
CREATE SPATIAL INDEX index name
  ON <object> ( spatial column name )
    {
      [ USING <geometry_grid_tessellation> ]
        WITH ( <bounding_box>
              [ [,] <tesselation_parameters> [ ,...n ] ]
              [ [,] <spatial_index_option> [ ,...n ] ] )
     | [ USING <geography_grid_tessellation> ]
         [ WITH ( [ <tesselation_parameters> [ ,...n ] ]
                [ [,] <spatial_index_option> [ ,...n ] ] ) ]
    }
  [ ON { filegroup_name | "default" } ]
```

其中的参数较多，这里就不逐一介绍，主要用到的几个参数如下所示。

- ❑ index_name：索引的名称。索引名称在表中必须唯一，但在数据库中不必唯一。索引名称必须符合标识符的规则。
- ❑ ON<对象>（spatial_column_name）：指定要对其创建索引的对象（数据库、架构或表）及空间列的名称。
- ❑ spatial_column_name：指定索引所基于的空间列。在一个空间索引定义中只能指定一个空间列，但是可以针对 geometry 或 geography 列创建多个空间索引。

- ❑ USING：指示空间索引的分割方案。此参数默认为特定于类型的值，geometry 对应的分割方案是 GEOMETRY_GRID，geography 对应的分割方案是 GEOGRAPHY_GRID。
- ❑ bounding_box：指定定义边界框 4 个坐标的一个数值 4 元组：左下角的最小 x 坐标和最小 y 坐标，以及右上角的最大 x 坐标和最大 y 坐标。
- ❑ GRIDS：定义分割方案中每一级别的网格密度。
- ❑ CELLS_PER_OBJECT=n：指定分割过程可以为索引中的单个空间对象使用的每个对象的分割单元格数：*n* 可以是介于 1～8192（包括 1 和 8192）之间的任何整数。每个对象的默认单元格数为 16。如果传递的数字无效或者该数字大于指定分割的最大单元格数，则会引发错误。

在顶层，如果对象包含的单元格多于 *n* 指定的单元格，则索引操作将根据需要使用尽可能多的单元格来提供完整的顶级分割。在这种情况下，对象收到的单元格数可能会大于指定的单元格数。此时，最大数即为顶级网格生成的单元格数，具体取决于密度。例如要对 GeometryTest 表中的 GeometryValue 列创建索引 SIndx_SpatialTable_geometry，对应的脚本如代码 15.71 所示。

代码 15.71　创建 Geometry 列的索引

```
CREATE SPATIAL INDEX SIndx_SpatialTable_geometry          --创建空间索引
   ON GeometryTest(GeometryValue)                         --表名和列名
   USING GEOMETRY GRID
   WITH (
    BOUNDING_BOX = ( 0, 0, 500, 200 ) ,                   --边界框
    GRIDS = (LOW, LOW, MEDIUM, HIGH),                     --网格密度
    CELLS_PER_OBJECT = 64,                                --分割成 64 个单元格
    PAD_INDEX  = ON --设置创建索引期间中间级别页中可用空间的百分比
    )
```

该索引指定了边界框为（0，0）～（500，200）之间的矩形。对于每一层分割的网格密度为：第 1、2 层 LOW，第 3 层为 MEDIUM，第 4 层为 HIGH。分割过程可以为索引中的单个空间对象使用的每个对象的分割单元格数为 64。

再如要在 GeographyTest 表中为 GeographyValue 列创建索引，对应的脚本如代码 15.72 所示。

代码 15.72　创建 Geography 列的索引

```
CREATE SPATIAL INDEX SIndx_SpatialTable_geography
   ON GeographyTest(GeographyValue)
   USING GEOGRAPHY GRID
   WITH (
    GRIDS = (MEDIUM, LOW, MEDIUM, HIGH ),    --网格密度
    CELLS_PER_OBJECT = 64,                   --分割成 64 个单元格
    PAD_INDEX  = ON ); --设置创建索引期间中间级别页中可用空间的百分比
```

该索引指定了每一层的网格密度为 MEDIUM，LOW，MEDIUM，HIGH。分割过程可以为索引中的单个空间对象使用的每个对象的分割单元格数为 64。

注意：对于地理网格索引，不能指定边界框。

15.4.3　使用 SSMS 创建空间索引

使用 SSMS 通过可视化的操作创建空间索引相对比较简单，例如，为 GeometryTest 表中的 GeometryValue 列创建空间索引的操作如下。

（1）在 SSMS 的对象资源管理器中展开 GeometryTest 表节点，选择“索引”节点，在弹出的快捷菜单中选择“新建索引”|“空间索引”选项，系统将打开“新建索引”对话框，如图 15.16 所示。

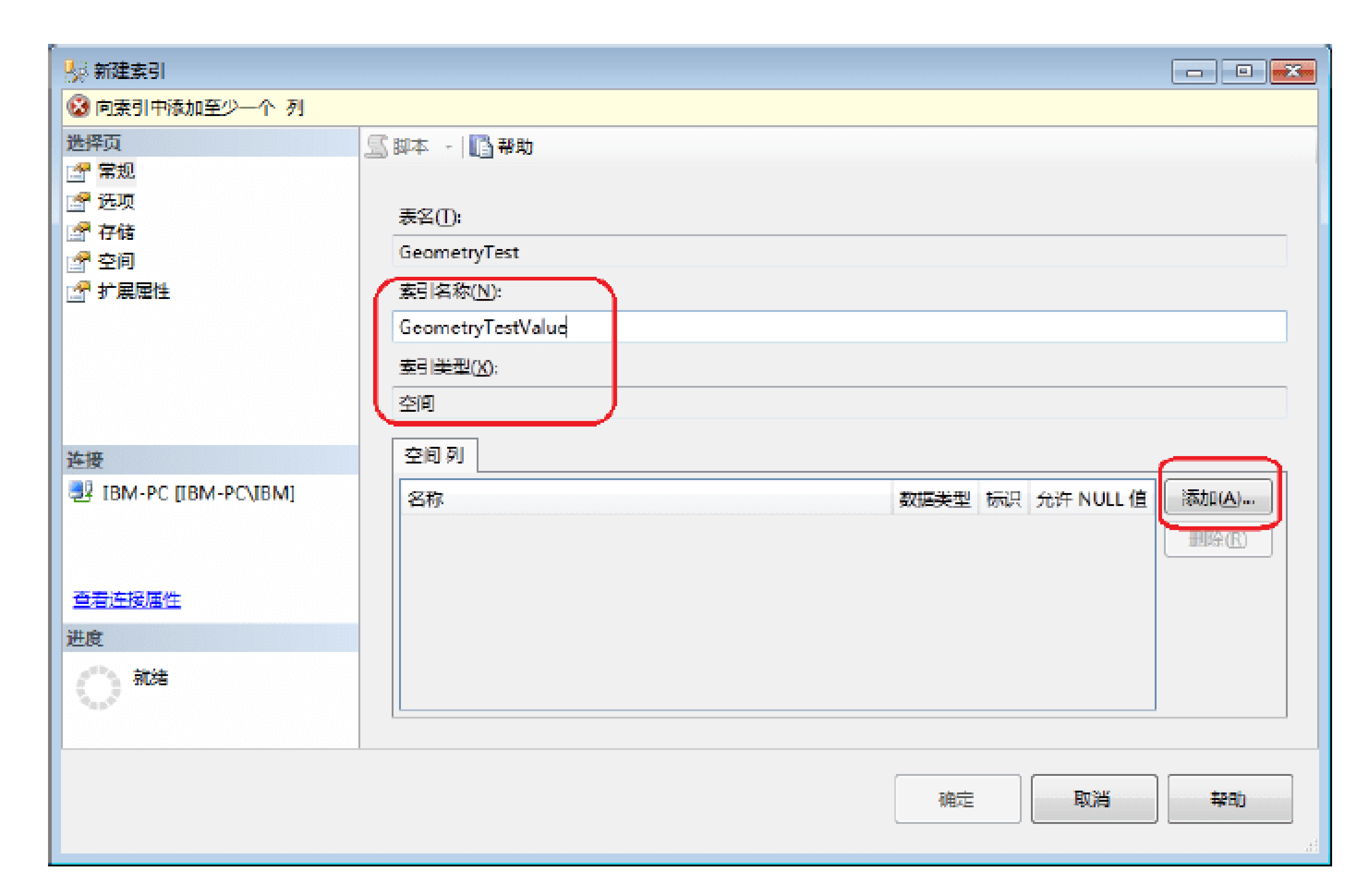


图 15.16　“新建索引”对话框

（2）在“索引名称”文本框中输入要新建的索引的名称 GeometryTestValue。

（3）单击“添加”按钮，将空间数据列 GeometryValue 添加到索引列中。

（4）选择“空间”选项页，系统切换到空间配置界面，如图 15.17 所示。

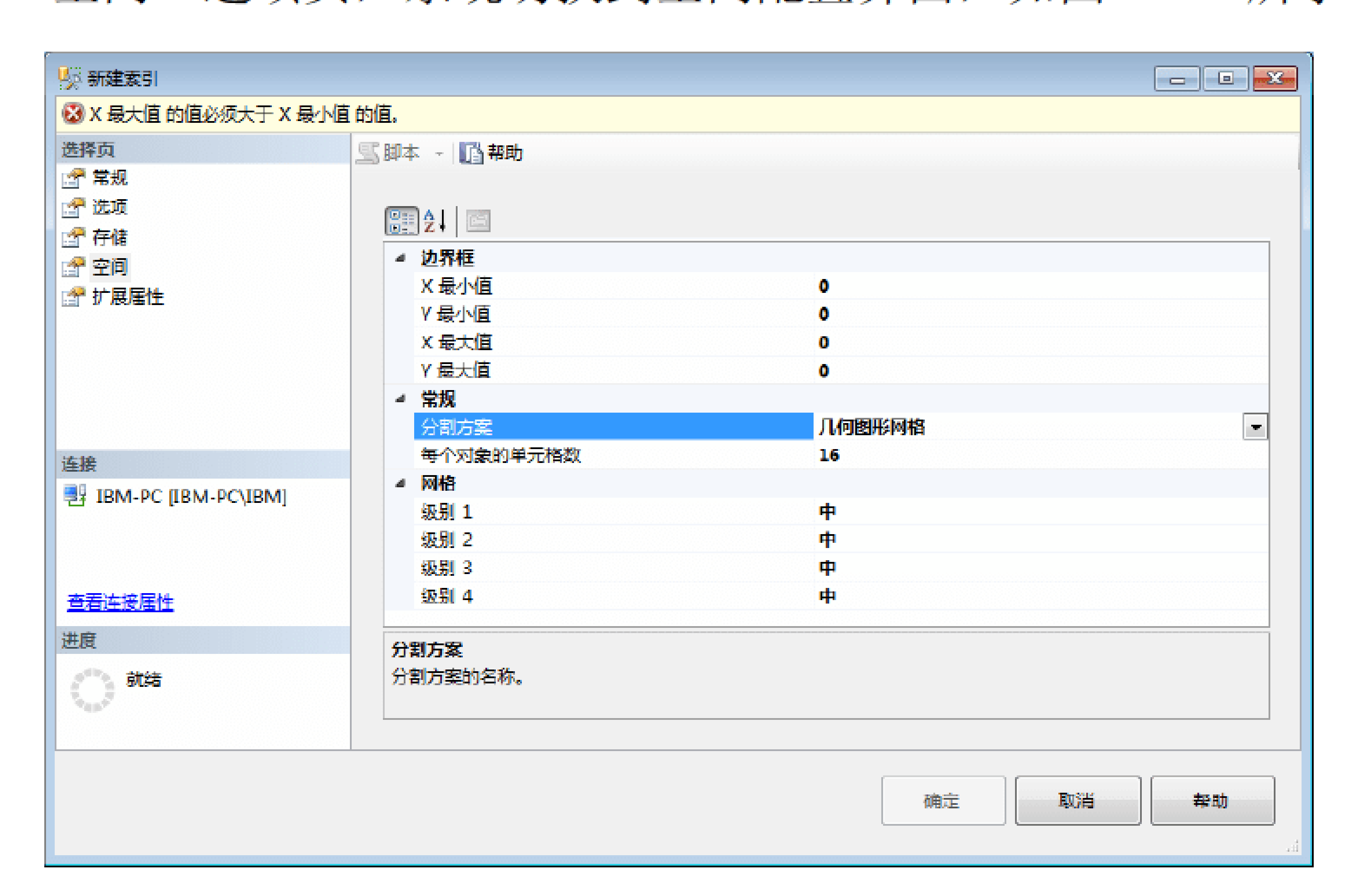


图 15.17　空间索引

（5）在该界面可以配置分割方案类型、边界框、每个级别的网格密度等。

（6）可以选择“选项”选项页切换到选项配置界面，如图 15.18 所示。

（7）根据需要修改索引选项，然后单击“确定”按钮，系统将创建空间索引。

对于 Geography 数据类型列，创建空间索引的方法与之相同。

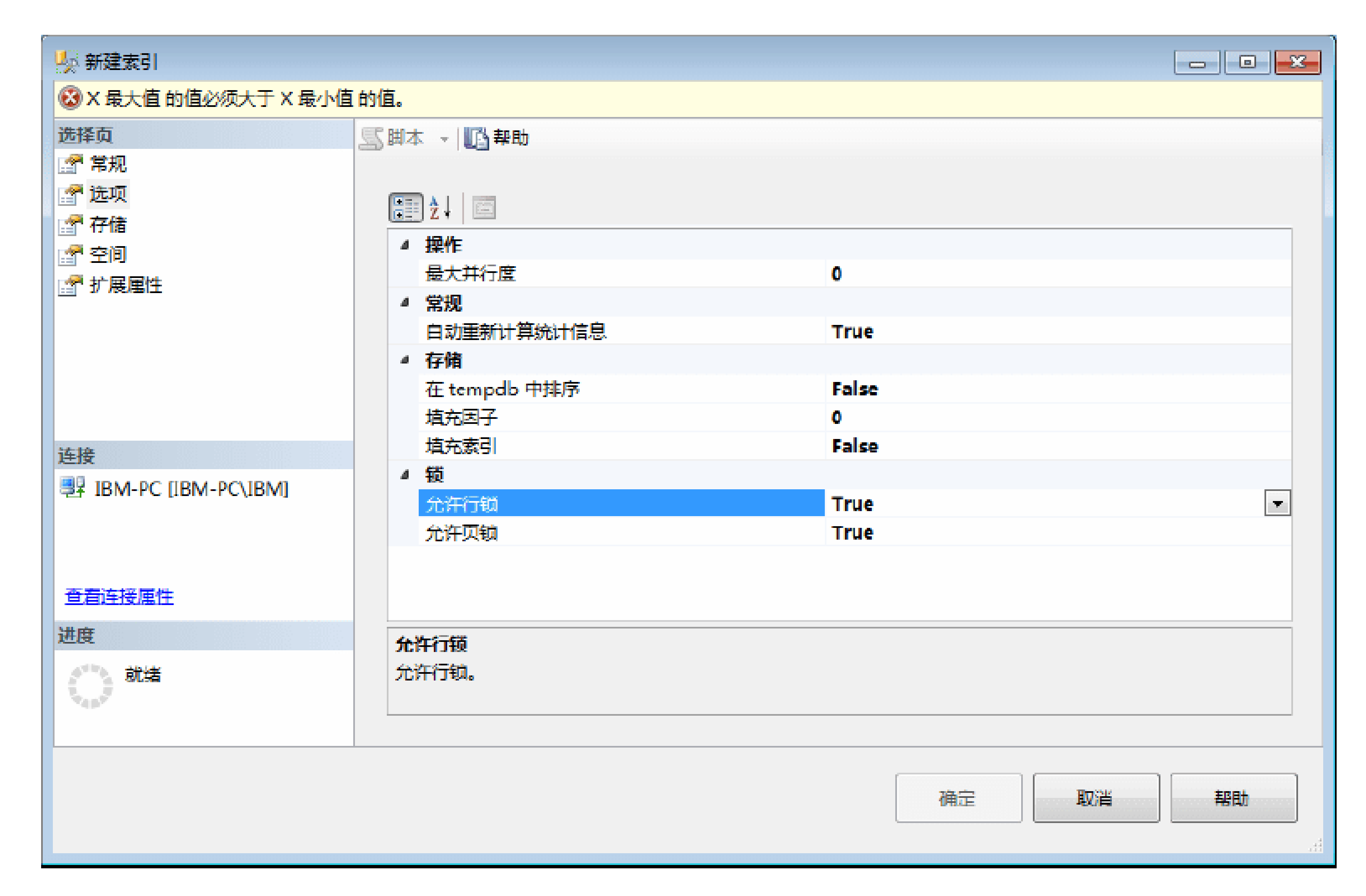


图 15.18　选项索引

15.4.4　管理空间索引

创建空间索引后若需要重建、禁用索引等，则使用 ALTER INDEX 命令。例如要禁用前面创建的空间索引 SIndx_SpatialTable_geometry，则对应的脚本如代码 15.73 所示。

代码 15.73　禁用索引

```
ALTER INDEX [SIndx_SpatialTable_geometry]
ON [dbo].[GeometryTest]
DISABLE  --禁用
```

禁用索引使用 Disable 选项，而重新启用索引使用 Rebuild 选项。

与一般的数据库对象不同的是，如果要修改空间索引的边界框值、分割方案、每个对象的单元格数和每个级别的网格密度时，不能使用 ALTER INDEX 命令，必须要先删除该索引再重新创建，或者直接使用 CREATE SPATIAL INDEX 命令跟上 DROP_EXISTING = ON 选项。

例如前面创建了空间索引 SIndx_SpatialTable_geometry，该索引中设置了边界框为 0，0，500，200。现在需要对该索引进行修改，将 200 改为 400，则对应的修改空间索引的脚本如代码 15.74 所示。

代码 15.74　修改空间索引

```
CREATE SPATIAL INDEX SIndx_SpatialTable_geometry
   ON GeometryTest(GeometryValue)
   USING GEOMETRY_GRID
   WITH (
     BOUNDING_BOX = ( 0, 0, 500, 400 ) ,
    GRIDS = (LOW, LOW, MEDIUM, HIGH),
    CELLS_PER_OBJECT = 64,
    PAD_INDEX  = ON ,
    DROP_EXISTING = ON  --该选项指定如果存在索引，则删除索引重新创建
    )
```

注意：跟上 DROP_EXISTING = ON 选项后不需要手动删除索引，系统将会在创建索引前自动删除。

若不需要再使用该索引，则可以将空间索引删除。删除空间索引使用 DROP INDEX 命令。删除空间索引的语法为：

```
DROP INDEX spatial_index_name ON spatial_table_name;
```

例如，要删除前面创建的空间索引 SIndx_SpatialTable_geometry，则对应的脚本为：

```
DROP INDEX SIndx_SpatialTable_geometry ON dbo.GeometryTest
```

使用 SSMS 修改和删除空间索引的方法与一般索引没有任何不同。修改索引需要调出索引的属性窗口进行修改，而删除索引使用快捷键 Delete 即可。

15.5　小　　结

本章主要讲解了 SQL Server 2012 数据类型——空间数据类型的相关知识。空间数据类型分为几何数据类型（geometry）和地理数据类型（geography）。SQL Server 中的空间数据类型根据 OGC 规范来使用。一般通过 WKT 熟知文本来创建空间数据类型，另外也可以通过 WKB 熟知二进制和 GML 来创建空间数据。SQL Server 中可以使用的空间实例类型包括：

- Point 单个点；
- MultiPoint 零个或多个点；
- LineString 单条线；
- MultiLineString 零个或多个线；
- Polygon 多边形；
- MultiPolygon 零个或多个多边形；
- GeometryCollection 各种空间类型的集合。

在构造了空间实例后可以获得实例的属性和方法，还可以获得实例之间的关系。

在构造空间实例时可以指定空间实例的 SRID，对于 geometry 数据类型 SRID 可以随便指定，而对于 geography 数据类型，SRID 必须是有效的值，可以通过查询 sys.spatial_reference_systems 目录视图找到有效的 SRID。不同的 SRID 的空间实例之间不能比较和运算。

“空间索引”是一种扩展索引，允许对空间列编制索引。空间列是包含空间数据类型（如 geometry 或 geography）数据的表列。

空间索引是将空间分割成网状层次结构。SQL Server 2012 中允许 4 层的层次结构。分割索引可以使用覆盖规则、每对象单元数规则和最深单元规则。

创建空间索引使用 CREATE SPATIAL INDEX 命令。而修改索引可以使用 ALTER INDEX 命令对基本索引选项进行修改，若要修改空间配置信息，则只能删除并新建索引。删除索引使用 DROP INDEX 命令。

第 16 章　跨实例链接

通过 SQL Server 2012 不仅可以访问本机实例的数据库，还可以通过跨实例链接的方式访问 Oracle、DB2、Sybase 等企业数据库和 Access、Excel 等文件数据库。本章将主要讲解跨实例链接的使用。

16.1　链接服务器

SQL Server 提供了链接服务器用于远程访问数据库对象。通过在链接服务器中配置访问接口，使用不同的接口访问不同的数据库类型，能够对企业内的异类数据源发出分布式查询、更新、命令和事务，并可以用相似的方式确定不同的数据源。

16.1.1　链接服务器简介

链接服务器定义指定了下列对象。

- OLE DB 访问接口，管理特定数据源和与其交互的 DLL。
- OLE DB 数据源，标识可通过 OLE DB 访问的特定数据库。

虽然通过链接服务器定义查询的数据源通常是数据库，但 OLE DB 访问接口对各种文件和文件格式可用。这些文件和文件格式包括文本文件、电子表格数据和全文内容搜索的结果。

链接服务器主要用于处理分布式查询。例如当客户端应用程序通过链接服务器执行分布式查询时，SQL Server 将分析命令，然后根据链接服务器链接的具体数据库生成请求命令，再向 OLE DB 发送请求。行集请求的形式可以是对该访问接口执行查询或从该访问接口打开基表。其查询数据的结构如图 16.1 所示。

为使数据源能通过链接服务器返回数据，该数据源的 OLE DB 访问接口（DLL）必须与 SQL Server 的实例位于同一服务器上。使用第三方 OLE DB 访问接口时，运行 SQL Server 服务的账户必须具有对安装访问接口的目录及其所有子目录的读取和执行权限。

微软默认为 SQL Server 提供了多个数据库的访问接口，访问 Access、Excel 或其他 SQL Server 实例时，不需要再使用第三方的 OLE DB 访问接口。但是对于 Sybase、Oracle 等数据库，若需要使用链接服务器，则需要在 SQL Server 服务器上安装第三方 OLE DB 访问接口。

16.1.2　使用 T-SQL 创建链接服务器

创建链接服务器需要使用 sp_addlinkedserver 系统存储过程。该存储过程的语法如代码 16.1 所示。

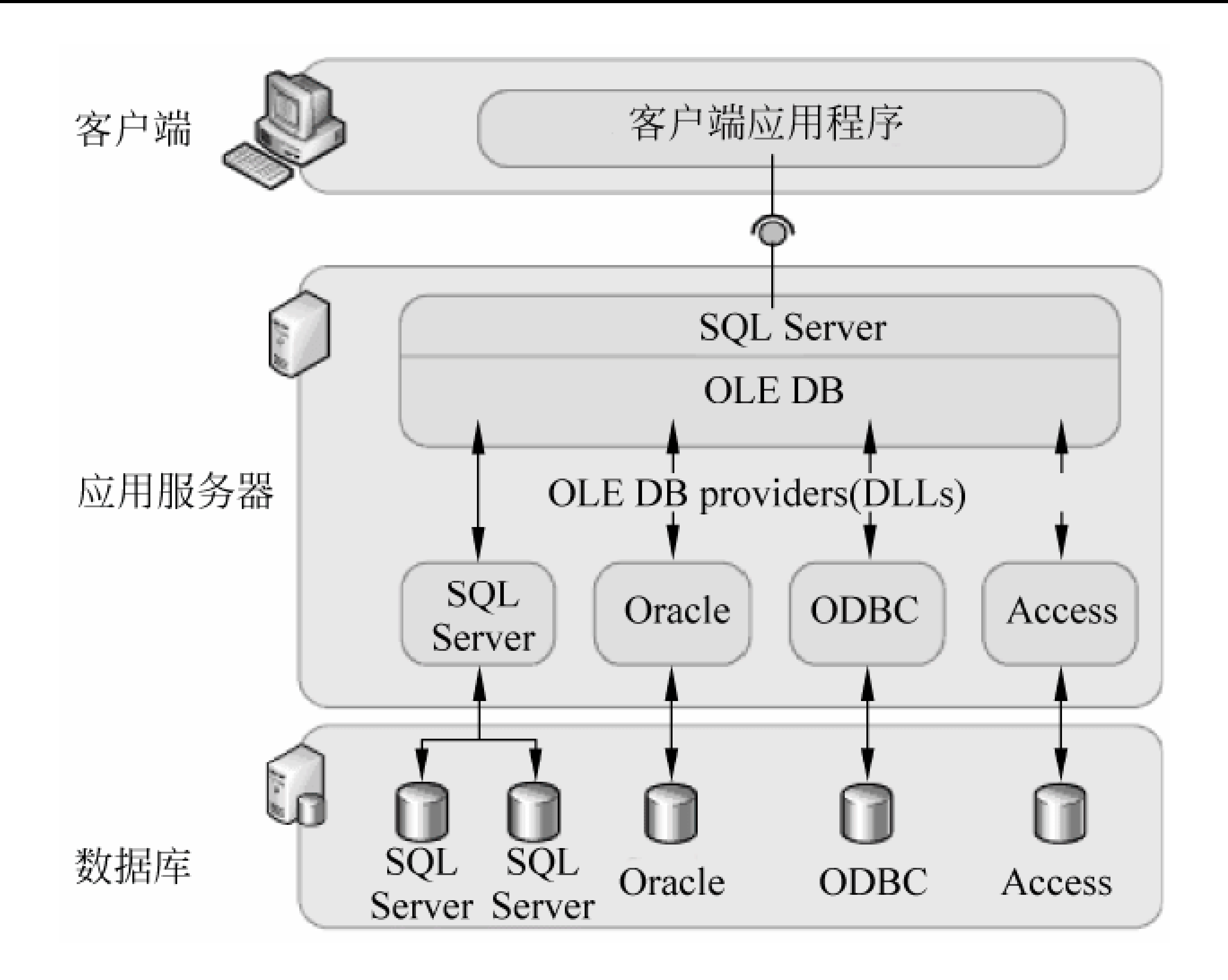


图 16.1　链接服务器访问数据的结构

代码 16.1　sp_addlinkedserver 语法

```
sp_addlinkedserver [ @server= ] 'server' [ , [ @srvproduct= ] 'product_name' ]
     [ , [ @provider= ] 'provider_name' ]
     [ , [ @datasrc= ] 'data_source' ]
     [ , [ @location= ] 'location' ]
     [ , [ @provstr= ] 'provider_string' ]
     [ , [ @catalog= ] 'catalog' ]
```

其中各个参数的含义如下。

- [@server=]'server'为要创建的链接服务器的名称。server 的数据类型为 sysname，没有默认值。
- [@srvproduct=]'product_name'为要添加为链接服务器的 OLE DB 数据源的产品名称。如果为 SQL Server，则不必指定 provider_name、data_source、location、provider_string 和 catalog。
- [@provider=]'provider_name'为与此数据源对应的 OLE DB 访问接口的唯一编程标识符（PROGID）。对于当前计算机中安装的指定 OLE DB 访问接口，provider_name 必须唯一。
- [@datasrc=]'data_source'为由 OLE DB 访问接口解释的数据源的名称。data_source 作为 DBPROP_INIT_DATASOURCE 属性传递以初始化 OLE DB 访问接口。
- [@location=]'location'为由 OLE DB 访问接口解释的数据库的位置。location 作为 DBPROP_INIT_LOCATION 属性传递以初始化 OLE DB 访问接口。
- [@provstr=]'provider_string'指定 OLE DB 访问接口特定的连接字符串，它可标识唯一的数据源。provstr 或传递给 IDataInitialize 或设置为 DBPROP_INIT_PROVIDER-STRING 属性以初始化 OLE DB 访问接口。
- [@catalog=]'catalog'为与 OLE DB 访问接口建立连接时所使用的目录。catalog 作为 DBPROP_INIT_CATALOG 属性传递以初始化 OLE DB 访问接口。在针对 SQL Server 实例定义链接服务器时，目录指向链接服务器映射到的默认数据库。

如表 16.1 所示为能通过 OLE DB 访问数据源而建立链接服务器的方法。对于特定的数

据源，可以使用多种方法为其设置链接服务器。该表中可能有多行适用于一种数据源类型。表 16.1 中还显示了用于设置链接服务器的 sp_addlinkedserver 参数值。

表 16.1　OLE DB接口配置

远程 OLE DB 数据源	OLE DB 访问接口	product_name	provider_name	data_source
SQL Server	Microsoft SQL Server Native Client OLE DB 访问接口	SQL Server（默认值）		
SQL Server	Microsoft SQL Server Native Client OLE DB 访问接口		SQLNCLI	SQL Server 的网络名称（用于默认实例）
SQL Server	Microsoft SQL Server Native Client OLE DB 访问接口		SQLNCLI	servername*instancename*（用于特定实例）
Oracle	Microsoft OLE DB Provider for Oracle	任何	MSDAORA	用于 Oracle 数据库的 SQL*Net 别名
Oracle，版本 8 及更高版本	Oracle Provider for OLE DB	任何	OraOLEDB.Oracle	用于 Oracle 数据库的别名
Access/Jet	Microsoft OLE DB Provider for Jet	任何	Microsoft.Jet.OLEDB.4.0	Jet 数据库文件的完整路径
ODBC 数据源	Microsoft OLE DB Provider for ODBC	任意	MSDASQL	ODBC 数据源的系统 DSN
ODBC 数据源	Microsoft OLE DB Provider for ODBC	任意	MSDASQL	
文件系统	Microsoft OLE DB Provider for Indexing Service	任意	MSIDXS	索引服务目录名称
Microsoft Excel 电子表格	Microsoft OLE DB Provider for Jet	任意	Microsoft.Jet.OLEDB.4.0	Excel 文件的完整路径
IBM DB2 数据库	Microsoft OLE DB Provider for DB2	任意	DB2OLEDB	

例如创建一个链接到 SQL Server 的链接服务器的脚本如代码 16.2 所示。

代码 16.2　创建链接服务器

```
EXEC master.dbo.sp_addlinkedserver  --新建链接服务器
@server = N'192.168.100.100',
@srvproduct=N'SQL Server'
```

再如要创建一个链接到 Access 数据库的链接服务器，其脚本如代码 16.3 所示。

代码 16.3　创建 Access 数据库的链接服务器

```
EXEC master.dbo.sp addlinkedserver
   @server = 'ACC', --链接服务器名
   @provider = 'Microsoft.Jet.OLEDB.4.0', --Provider
```

```
    @srvproduct = 'OLE DB Provider for Jet',--驱动
    @datasrc = 'D:\Access1.mdb'  --Access 数据库文件地址
```

创建的链接服务器若需要用户认证才能访问，则需要使用 sp_addlinkedsrvlogin 系统存储过程为链接服务器添加用户认证信息，其语法如代码 16.4 所示。

代码 16.4　sp_addlinkedsrvlogin 语法

```
sp addlinkedsrvlogin [ @rmtsrvname = ] 'rmtsrvname'
    [ , [ @useself = ] 'TRUE' | 'FALSE' | 'NULL']
    [ , [ @locallogin = ] 'locallogin' ]
    [ , [ @rmtuser = ] 'rmtuser' ]
    [ , [ @rmtpassword = ] 'rmtpassword' ]
```

其中各个参数的含义如下。

- [@rmtsrvname=]'rmtsrvname'为应用登录映射的链接服务器的名称。
- [@useself=]'TRUE'|'FALSE'|'NULL'确定是否通过模拟本地登录名，或显式提交登录名和密码连接到 rmtsrvname。值为 TRUE 指定登录名使用自己的凭据连接到 rmtsrvname，而忽略 rmtuser 和 rmtpassword 参数。值为 FALSE 指定 rmtuser 和 rmtpassword 参数用于连接到指定 locallogin 的 rmtsrvname。如果 rmtuser 和 rmtpassword 也设置为 NULL，则不使用登录名或密码连接链接服务器。
- [@locallogin=]'locallogin'为本地服务器上的登录。NULL 指定此项应用于连接到 rmtsrvname 的所有本地登录。如果不为 NULL，则 locallogin 可以是 SQL Server 登录或 Windows 登录。对于 Windows 登录来说，必须以直接的方式或通过已被授权访问的 Windows 组成员身份授予其访问 SQL Server 的权限。
- [@rmtuser=]'rmtuser'为当@useself 为 FALSE 时，用于连接到 rmtsrvname 的远程登录名。当远程服务器不使用 Windows 身份验证的 SQL Server 实例时，rmtuser 是一个 SQL Server 登录名。rmtuser 的数据类型为 sysname，默认值为 NULL。
- [@rmtpassword=]'rmtpassword'为与 rmtuser 关联的密码。rmtpassword 的数据类型为 sysname，默认值为 NULL。

例如，对于前面创建的链接服务器 192.168.100.100，为其添加用户认证信息的脚本如代码 16.5 所示。

代码 16.5　添加链接服务器的用户认证

```
EXEC master.dbo.sp_addlinkedsrvlogin      --添加链接服务器的用户认证
@rmtsrvname=N'192.168.100.100',           --链接服务器名
@useself=N'False',                        --不模拟本地用户登录
@locallogin=NULL,                         --本地服务器登录
@rmtuser=N'sa',                           --用户名
@rmtpassword='p@ssw0rd'                   --密码
```

当下列所有条件都存在时，SQL Server 可以自动使用正在发出查询用户的 Windows 安全凭据（Windows 登录名和密码），以连接到链接服务器，而不必使用 sp_addlinkedsrvlogin 创建一个预设的登录映射。

- 使用 Windows 身份验证模式，用户连接到 SQL Server。
- 在客户端和发送服务器上安全账户委托是可用的。

- ❑ 提供程序支持 Windows 身份验证模式（例如，运行于 Windows 上的 SQL Server）。

16.1.3　使用 SSMS 创建链接服务器

使用 SSMS 创建链接服务器的主要操作如下。

（1）在 SSMS 的对象资源管理器中展开“服务器对象”节点，选择其下的“链接服务器”子节点，在弹出的快捷菜单中选择“新建链接服务器”选项，弹出“新建链接服务器”对话框，如图 16.2 所示。

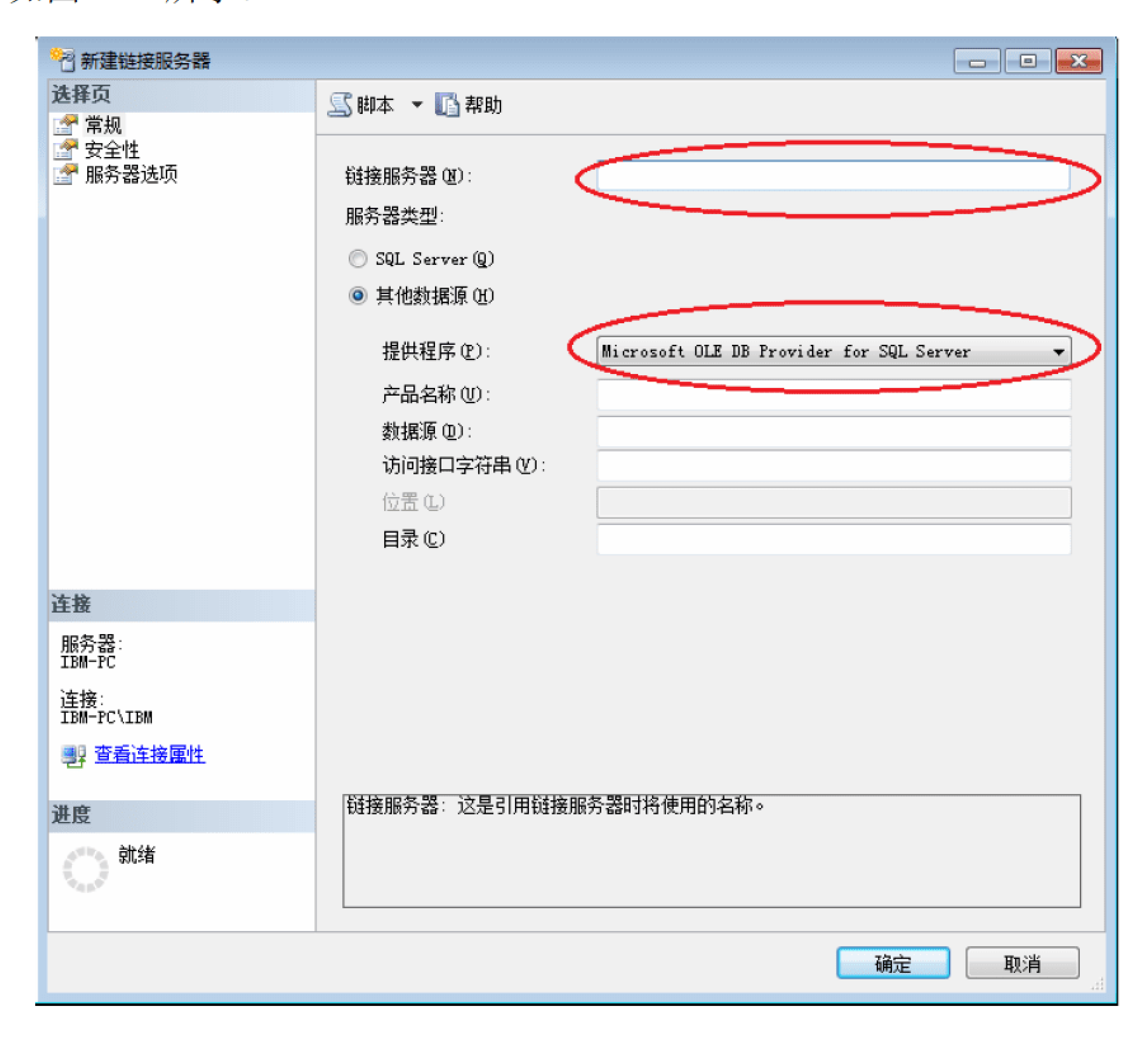


图 16.2　“新建链接服务器”对话框

（2）在“链接服务器”文本框中输入要新建的链接服务器的名称，在“提供程序”下拉列表框中选择需要使用的接口，然后分别在“产品名称”、“数据源”、“访问接口字符串”文本框中输入产品名称、数据源和访问接口字符串。

（3）选择“安全性”选项页，系统切换到安全配置界面，如图 16.3 所示。

（4）SQL Server 支持为每个不同的登录用户映射不同的链接服务器认证。可以单击“添加”按钮将本地登录的用户与远程用户映射起来。对于没有映射的用户，SQL Server 提供了以下 4 种方案：

- ❑ 不建立连接。
- ❑ 不使用安全上下文建立连接。
- ❑ 使用登录名的当前安全上下文建立连接。
- ❑ 使用其他配置的安全上下文建立连接。

这里若希望没有指定映射的用户不能连接链接服务器，则可以选择“不建立连接”单选按钮。

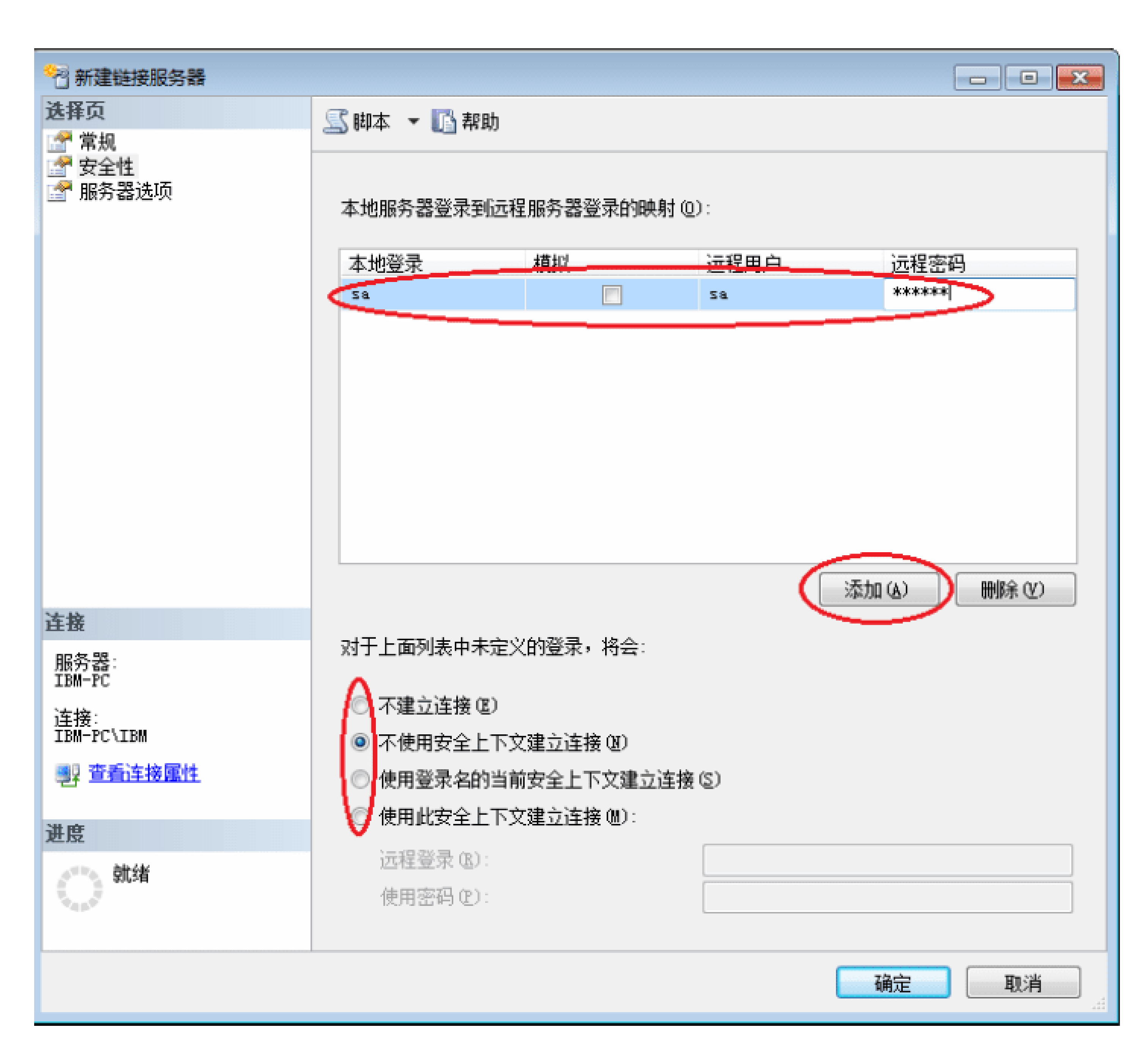


图 16.3　链接服务器安全性配置

（5）选择“服务器选项”选项页，系统进入链接服务器其他选项配置界面，如图 16.4 所示。

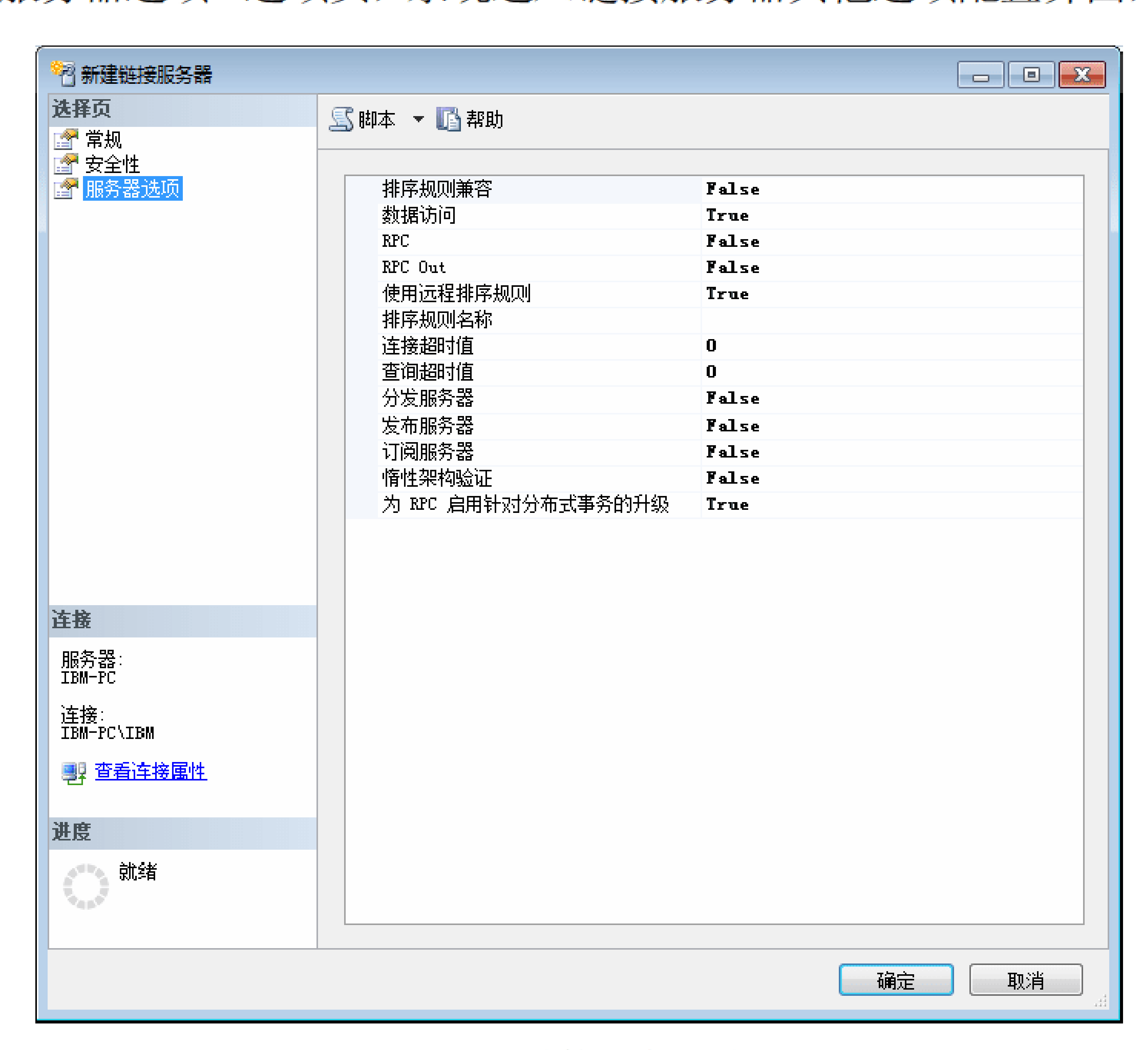


图 16.4　链接服务器选项

（6）在服务器选项中可以配置是否允许调用链接服务器上的存储过程、排序规则、连接超时时间等。

（7）单击“确定”按钮，系统完成链接服务器的创建。

16.1.4　修改链接服务器属性

链接服务器器在创建后不可修改其驱动、名字等。简单来说就是在 sp_addlinkedserver 存储过程中使用的这些参数，在创建链接服务器后不可修改。若需要修改只有删除原有链接服务器后再重新建立新的链接服务器。

但是链接服务器的安全配置和服务器选项是可以修改的。修改链接服务器的安全性仍然使用 sp_addlinkedsrvlogin 存储过程。对于本地用户映射远程用户的修改，则需要使用 sp_droplinkedsrvlogin 存储过程删除原有用户映射。

删除链接服务器指定登录格式为：

```
sp droplinkedsrvlogin [ @rmtsrvname= ] 'rmtsrvname' ,
   [ @locallogin= ] 'locallogin'
```

其中的参数[@rmtsrvname=]'rmtsrvname'是应用 SQL Server 登录映射的链接服务器名称。rmtsrvname 的数据类型为 sysname，无默认值。rmtsrvname 必须已经存在。

[@locallogin=]'locallogin'表示当前服务器上映射到链接服务器的 SQL Server 登录，它具有到链接服务器 rmtsrvname 的映射。locallogin 的数据类型为 sysname，无默认值。locallogin 到 rmtsrvname 的映射必须已经存在。如果为 NULL，则删除 sp_addlinkedserver 创建的默认映射（该映射将本地服务器上的所有登录映射到链接服务器上的登录）。

例如将原有的 sa 用户映射删除而使用新的登录名 sa2 进行用户映射，对应的脚本如代码 16.6 所示。

代码 16.6　删除和增加映射用户

```
EXEC master.dbo.sp droplinkedsrvlogin
@rmtsrvname = N'192.168.100.100',
@locallogin = N'sa'
--重新增加用户
EXEC master.dbo.sp_addlinkedsrvlogin
@rmtsrvname = N'192.168.100.100',
@locallogin = N'sa2', --新的登录名
@useself = N'False',
@rmtuser = N'',
@rmtpassword = N'123'
```

而修改链接服务器属性需要使用系统存储过程 sp_serveroption，该存储过程的语法如代码 16.7 所示。

代码 16.7　sp_serveroption 语法格式

```
sp_serveroption [@server = ] 'server'
     ,[@optname = ] 'option_name'
     ,[@optvalue = ] 'option_value' ;
```

其中的参数[@server=]'server'是要为其设置选项的服务器的名称。server 的数据类型为 sysname，没有默认值。[@optname=]'option_name' 为指定的服务器设置的选项。[@optvalue=]'option_value'指定应启用（TRUE 或 on）还是禁用（FALSE 或 off）option_name。option_value 可以是用于 connecttimeout 和 querytimeout 选项的非负整数。对于 collationname

选项，option_value 可以是排序规则名称或 NULL。

例如要将链接服务器设置为可以远程过程调用，则对应的修改如代码 16.8 所示。

代码 16.8　修改链接服务器选项

```
EXEC master.dbo.sp serveroption
@server=N'192.168.100.100',
@optname=N'rpc', --可以远程过程调用
@optvalue=N'true'
```

如果不需要使用链接服务器，需要将其删除，可以使用 sp_dropserver 系统存储过程，其语法格式为：

```
sp dropserver [ @server = ] 'server'
    [ , [ @droplogins = ] { 'droplogins' | NULL} ]
```

其中的参数[@server=]'server'是要删除的服务器。server 的数据类型为 sysname，无默认值。server 必须存在。

[@droplogins=]'droplogins'|NULL 为如果指定了 droplogins，那么对于 server，还必须删除相关的远程服务器和链接服务器登录名。@droplogins 的数据类型为 char(10)，默认值为 NULL。例如要删除前面创建的链接服务器，对应的脚本如代码 16.9 所示。

代码 16.9　删除链接服务器

```
EXEC master.dbo.sp dropserver
@server=N'192.168.100.100', --删除指定链接服务器
@droplogins='droplogins'
```

关于使用 SSMS 修改和删除链接服务器的操作，与修改和删除其他数据库对象的操作并没有什么不同，笔者这里不再重述。

16.1.5　使用链接服务器

以前面创建的链接到 Access 数据库的链接服务器 ACC 为例，现在 Access 数据库中创建一个表 Tb1 和添加一些数据，然后在 SSMS 的对象资源管理器中可以展开 ACC 链接服务器节点，直到看见其中的表 Tb1，如图 16.5 所示。

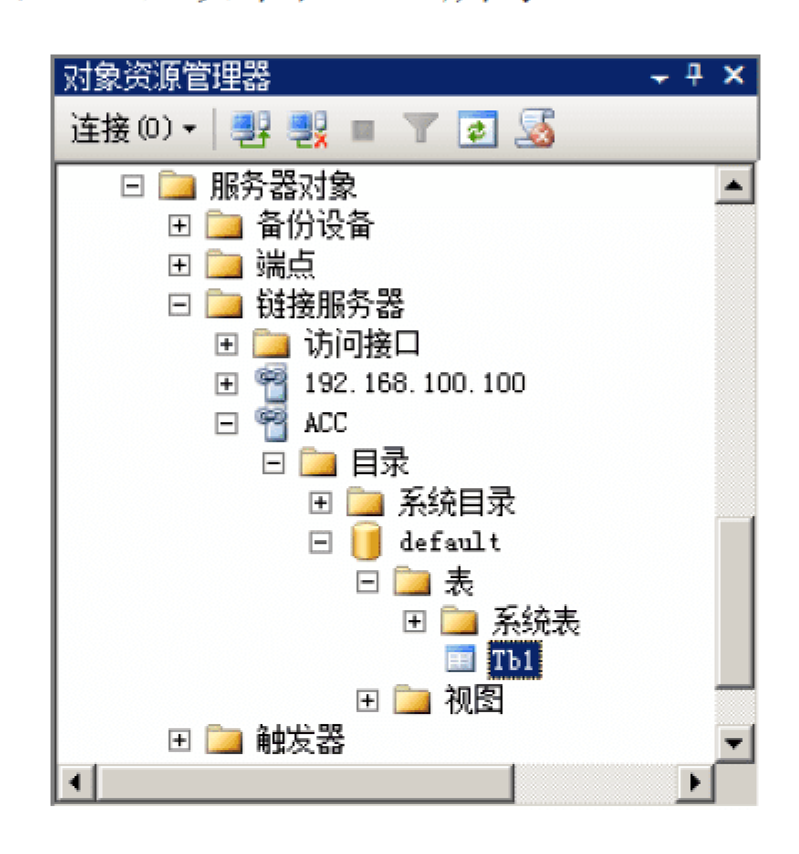


图 16.5　查看链接服务器中的表

若要查询该表，选择 Tb1 选项，在弹出的快捷菜单中选择“编写表脚本为”|“SELECT

到”|“新查询编辑器窗口”命令，系统将为该表的查询编写脚本如代码 16.10 所示。

代码 16.10　查询链接服务器中的表

```
SELECT [ID]
      ,[StuName]
      ,[ClassID]
      ,[Birthday]
      ,[Sex]
  FROM [ACC]...[Tb1]     --ACC 为链接服务器名
```

运行该脚本便可查询出 Access 数据库中该表的数据。

对于不同的数据库类型，通过链接服务器查询数据的方式和要求是不同的。FROM 子句后跟的是数据库对象全名。数据库对象的全名使用如下格式表示：

```
[服务器名].[数据库名].[架构名].[对象名]
```

这里由于 Access 数据库不存在数据库名和架构名，所以省略后就变成了[ACC]...[Tb1]。

对于 Oracle 数据库的链接服务器，由于不存在数据库名，所以 Oracle 链接服务器中数据库名将省略。Oracle 数据库的对象全部是使用大写表示的，所以在 Oracle 数据库链接服务器中不能出现小写字母。

虽然链接服务器主要是用于分布式查询，但是仍然可以通过链接服务器更新远程数据库中的数据。例如要向 ACC 链接服务器中添加一行记录，则对应的脚本如代码 16.11 所示。

代码 16.11　向链接服务器中添加数据

```
INSERT INTO  [ACC]...[Tb1]  --插入数据到链接服务器
([ID]
,[StuName]
,[ClassID]
,[Birthday]
,[Sex])
VALUES (2,'章林林',1,'1984-11-1',0)
```

16.2　同　义　词

同义词是架构范围内的对象的另一名称。通过使用同义词，客户端应用程序可以使用由一部分组成的名称来引用基对象，而不必使用由两部分、三部分或四部分组成的名称。

16.2.1　同义词简介

如果要查询链接服务器中的一个对象，例如 AdventureWorks2012 数据库中的 AddressType 表，则 FROM 子句后将要跟如下的对象：

```
[192.168.100.100].[AdventureWorks2012].[Person].[AddressType]
```

为了简化输入，所以在 SQL Server 中使用了同义词。同义词是用来实现下列用途的数据库对象。

- 为可以存在于本地或远程服务器上的其他数据库对象（称为基对象）提供备用名称。
- 提供抽象层以避免用户对客户端应用程序基对象的名称或位置进行更改。

同义词从属于架构，并且与架构中的其他对象一样，其名称必须是唯一的。可以为下列数据库对象创建同义词：CLR 存储过程、CLR 表值函数、CLR 标量函数、CLR 聚合函数、复制筛选过程、扩展存储过程、SQL 标量函数、SQL 表值函数、SQL 内联表值函数、SQL 存储过程、视图、表。

一个同义词不能是另一个同义词的基对象，也不能引用用户定义聚合函数。同义词与其基对象之间只是按名称绑定。对基对象的存在性、类型和权限检查都在运行时执行。因此，可以修改或删除基对象，也可以先删除基对象，然后用与原始基对象同名的另一个对象替换该基对象。例如，有一个引用 AdventureWorks2012 中的 Person.Contact 表的同义词 MyContacts，如果将 Contact 表删除，并用名为 Person.Contact 的视图替换该表，则 MyContacts 将引用 Person.Contact 视图。

同义词的引用不是绑定到架构的，因此，可以随时删除同义词。但删除同义词后，会留下已删除同义词的无关联引用，而只有在运行时才会发现这些引用。

16.2.2　创建同义词

创建同义词使用 CREATE SYNONYM 命令，其语法格式如代码 16.12 所示。

代码 16.12　创建同义词语法

```
CREATE SYNONYM [ schema name 1. ] synonym name FOR < object >
< object > :: =
{
    [ server name.[ database name ] . [ schema name 2 ].| database name .
    [ schema name 2 ].| schema name 2. ] object name
}
```

其中各个参数的含义如下。

- schema_name_1：指定创建同义词所使用的架构。如果未指定 schema，SQL Server，将使用当前用户的默认架构。
- synonym_name：新同义词的名称。
- server_name：基对象所在服务器的名称。
- database_name：基对象所在数据库的名称。如果未指定 database_name，则使用当前数据库的名称。
- schema_name_2：基对象架构的名称。如果未指定 schema_name，则使用当前用户的默认架构。
- object_name：同义词被引用基对象的名称。

🔔注意：创建同义词时不需要基对象存在。SQL Server 将在运行时检查基对象是否存在。

例如，为链接服务器中的 AdventureWorks2012 数据库中的 AddressType 表创建同义词，

对应的脚本如代码 16.13 所示。

代码 16.13　创建同义词

```
USE tempdb;
GO
CREATE SYNONYM AddressType --创建同义词
FOR [192.168.100.100].[AdventureWorks].[Person].[AddressType]
```

除了可以给链接服务器中的数据库对象创建同义词外，也可以给本实例中的数据库对象创建同义词。例如，在 tempdb 中为 AdventureWorks2012 数据库的 Person.Address 表创建同义词的脚本如代码 16.14 所示。

代码 16.14　为本地服务器创建同义词

```
USE tempdb;
GO
CREATE SYNONYM MyAddress
FOR AdventureWorks.Person.Address;
```

16.2.3　使用同义词

在创建好同义词后，便可将同义词作为一般的数据库对象来访问。例如要访问读取链接服务器中的数据，则对应的脚本如代码 16.15 所示。

代码 16.15　通过同义词访问链接服务器

```
USE tempdb;
GO
SELECT *
FROM AddressType
```

同义词还可以用于存储过程、函数等。例如，创建一个链接服务器中一个存储过程的同义词，并调用该存储过程的脚本如代码 16.16 所示。

代码 16.16　通过同义词调用存储过程

```
USE tempdb;
GO
CREATE SYNONYM GetDeparment     --存储过程的同义词
FOR [192.168.100.100].[AdventureWorks2012].[dbo].[GetDeparment]
GO
EXEC GetDeparment               --执行同义词
```

注意：如果链接服务器没有打开 RPC，则无法通过链接服务器执行存储过程。

不使用同义词后，可以使用 DROP SYNONYM 命令将其删除。删除同义词的语法为：

```
DROP SYNONYM [ schema. ] synonym_name
```

例如要将同义词 GetDeparment 删除，则对应的脚本如代码 16.17 所示。

代码 16.17　删除同义词

```
USE tempdb;
GO
DROP SYNONYM GetDeparment
```

16.3　深入探讨跨实例链接

在了解了链接服务器的设置和同义词之后，本节主要深入探讨通过链接服务器进行其他跨实例的链接。

16.3.1　数据查询方式

在前面介绍复制的章节中，笔者介绍了 OPENROWSET()函数可以用于将远程服务器中的数据查询出来，进行批量数据导入。对于在建立链接服务器的情况下使用 OPENROWSET()函数进行查询是个不错的选择，但是如果建立了链接服务器，则可以通过链接服务器或同义词访问远程数据库。

虽然直接查询链接服务器可以获得查询结果，但是对于 Oracle 等数据库的链接服务器来说，查询并不都运行在 Oracle 服务器上。例如现有一个 Oracle 链接服务器 ORA，要查询一个几十万甚至上百万条记录的表 ORDERS 中的一个数据，则通过链接服务器查询的脚本如代码 16.18 所示。

代码 16.18　查询 Oracle 数据库中的数据

```
SELECT *
FROM ORA..MARY.ORDERS
WHERE ORDERCODE='20080808008'
```

在 SQL Server 中运行该脚本可能要等上 10 秒、20 秒甚至 1 分钟、5 分钟才可能查询出结果。但是如果将脚本在 Oracle 服务器上直接运行，则 1 秒钟不到就查询出结果了。造成这种情况的是 SQL Server 查询链接服务器的机制。

不同的数据库对应的 SQL 语言有所不同。而对于 Oracle 数据库，通过链接服务器查询数据时，SQL Server 为了保证 T-SQL 语句能够正常使用，但是 Oracle 数据库可能不认识这些 T-SQL 语句，所以 SQL Server 将会把查询中所用到的 Oracle 表数据从 Oracle 数据库中读出来，一直到满足查询条件为止。对于代码 16.18 中的查询，SQL Server 会将 Oracle 数据库中的 ORDERS 表全部读取到 SQL Server 数据库中，一边读取一边查找 ORDERCODE= '20080808008'的数据，直到全部数据读取完为止。对于上十万、百万级的数据表来说，全部读取数据当然会造成系统缓慢。

SQL Server 为了解决这个问题，提供了 OPENQUERY()函数用于将查询语句直接送到链接服务器中，由链接服务器的数据库引擎负责查询，而不是由 SQL Server 将全部数据读取到本地进行查询。OPENQUERY()函数的语法格式为：

```
OPENQUERY ( linked_server ,'query' )
```

其中，linked_server 为表示链接服务器名称的标识符。query 为在链接服务器中执行的查询字符串，该字符串的最大长度为 8KB。对于代码 16.18 中的查询，修改为 OPENQUERY()函数的脚本如代码 16.19 所示。

代码 16.19　使用 OPENQUERY()函数查询链接服务器

```
SELECT *
FROM OPENQUERY(ORA, --ORA 为 Oracle 的链接服务器
'SELECT * FROM MARY.ORDERS WHERE ORDERCODE=''20080808008''') --Oracle 查询
```

使用 OPENQUERY()函数后，Oracle 数据库将负责查询，只将查询的结果返回给 SQL Server。

16.3.2　链接服务器的安全

在链接服务器的连接过程（如处理分布式查询时）中，发送服务器提供登录名和密码以代表自己连接到接收服务器。为了使该连接有效，必须使用 SQL Server 系统存储过程在链接服务器之间创建登录名映射。

前面已经说到可以使用 sp_addlinkedsrvlogin 添加链接服务器的用户映射，使用 sp_droplinkedsrvlogin 进行删除。链接服务器登录名映射为指定的链接服务器和本地登录名建立远程登录名和密码。在 SQL Server 连接到链接服务器以执行分布式查询或存储过程时，SQL Server 将查找正在执行查询或存储过程的当前登录名的登录名映射。如果有一个登录名映射，则 SQL Server 在连接到链接服务器时将发送相应的远程登录名和密码。

例如，使用远程密码 p@ssw0rd 为链接服务器 S1 建立了一个从本地登录名 U1 到远程登录名 U2 的映射。本地登录名 U1 执行访问链接服务器 S1 中存储的表的分布式查询过程中，当 SQL Server 连接到链接服务器 S1 时，将 U2 和 p@ssw0rd 作为用户 ID 和密码进行传递。

链接服务器配置的默认映射模拟了登录名的当前安全凭据，这类映射称为自映射。如果使用 sp_addlinkedserver 添加链接服务器，则将为所有本地登录名添加默认自映射。如果安全账户委托可用，且链接服务器支持 Windows 身份验证，则支持经过 Windows 身份验证的登录名的自映射。

注意：请尽可能使用 Windows 身份验证。

如果在客户端或发送服务器上安全账户委托不可用，或链接服务器/访问接口不能识别 Windows 身份验证模式，则自映射对使用 Windows 身份验证的登录名不起作用。因此，必须建立一个从使用 Windows 身份验证的登录名，到链接服务器上不使用 Windows 身份验证的特定登录名的本地登录名映射。在这种情况下，如果链接服务器是 SQL Server 实例，则远程登录名使用 SQL Server 身份验证。

分布式查询受限于链接服务器授予远程登录名对远程表的权限，但是 SQL Server 在编译时不执行任何权限验证。访问接口在查询执行时检测并报告任何违反权限的行为。

16.3.3　目录服务

目录服务（Microsoft Windows Active Directory，活动目录，简称 AD）是微软为企业环境设计的统一用户权限的平台。微软的大部分企业级产品比如 Excange、Biztalk、SQL Server、MOSS 等都可以集成到 AD 中。通过 SQL Server 链接服务器可以实现对目录服务的访问。

Microsoft 目录服务的 Microsoft OLE DB 访问接口，可以访问 Microsoft Windows 2000 目录服务中的信息。使用此访问接口的查询可返回对象的最大数量为 1000。例如创建目录服务的链接服务器的脚本如代码 16.20 所示。

代码 16.20　创建目录服务的链接服务器

```
EXEC sp addlinkedserver 'ADSI', 'Active Directory Services 2.5',
'ADSDSOObject', 'adsdatasource'  --创建目录服务链接服务器
GO
```

对于 Windows 验证登录，只需自映射就足以通过使用 SQL Server 安全委托访问目录。因为默认情况下会为通过运行 sp_addlinkedserver 创建的链接服务器创建自映射，所以不需要其他登录映射。

用于 Microsoft 目录服务的 Microsoft OLE DB 访问接口支持两种命令方言（LDAP 和 SQL）查询目录服务。可以使用 OPENQUERY()函数将命令发送到目录服务，并在 SELECT 语句中使用其结果。

例如创建一个视图，它使用 OPENQUERY()返回域地址为 sales.adventure-works.com 的服务器 ADSISrv 上目录中的信息。OPENQUERY()函数中的命令是一个针对目录的 SQL 查询，该查询返回对象的 Name、SN 和 ST 属性，这些对象属于目录中指定的层次结构位置（OU=Sales）上的 contact 类。因而，可在任何 SQL Server 查询中使用该视图。具体脚本如代码 16.21 所示。

代码 16.21　查询目录服务

```
CREATE VIEW viewADContacts           --定义视图
AS
SELECT [Name], SN [Last Name], ST State
FROM OPENQUERY( ADSI,                --ADSI 为目录服务器链接服务器
   'SELECT Name, SN, ST
   FROM ''LDAP://ADSISrv/ OU=Sales,DC=sales,DC=adventure-works,DC=com''
   WHERE objectCategory = ''Person'' AND
      objectClass = ''contact''')    --查询目录服务器
GO
SELECT * FROM viewADContacts
```

16.3.4　索引服务

Windows Server 2008 包括 IIS 和 Microsoft 索引服务，这些服务基于文件的属性启用筛选文件，并执行全文索引和文件数据的检索。

索引服务还包括用于 Microsoft 索引服务的 Microsoft OLE DB 访问接口。此访问接口可用来对非数据库文件进行全文索引或属性值搜索。可使用 sp_addlinkedserver 完成链接服务器定义，然后，分布式查询可引用访问接口来检索索引信息。例如要创建一个链接服务器以访问索引服务全文索引，具体步骤如下。

（1）使用索引服务创建全文索引。默认情况下，索引服务安装一个名为 default 的目录。

（2）执行 sp_addlinkedserver 以创建链接服务器，指定 MSIDXS 为 provider_name 和全文索引的名称为 data_source。

例如，若要创建访问全文索引（名为 Web）的链接服务器（名为 FTIndexWeb），请执行：

```
sp_addlinkedserver 'FTIndexWeb', 'Index Server', 'MSIDXS', 'Web'
```

（3）索引服务客户端的安全授权，是根据使用 Microsoft 索引服务的 OLE DB 访问接口进程的 Windows 账户进行的。对于 SQL Server 经过身份验证的登录名，分布式查询在 SQL Server 进程的上下文中运行。由于 SQL Server 通常在拥有高级授权的账户下运行，因此某些 SQL Server 的经过身份验证的用户，也可以使用索引服务链接服务器访问未经授权访问的信息。sysadmin 固定服务器角色成员通过严格控制，可以使用索引服务链接服务器执行分布式查询的 SQL Server 登录名解决这一问题。

管理员首先使用 sp_droplinkedsrvlogin，删除映射到索引服务链接服务器的所有登录名，例如：

```
sp_droplinkedsrvlogin FTIndexWeb, NULL
```

然后管理员使用 sp_addlinkedsrvlogin 为单个登录名授予访问链接服务器的权限，例如：

```
sp_addlinkedsrvlogin FTIndexWeb, true, 'SomeLogin'
```

Transact-SQL 语句可以使用 OPENQUERY()函数向索引服务发出命令，使用的 SQL 语法与 SQL Server 支持的全文查询语法一致，后者是对数据库所存储数据进行全文检索的语法。

16.4　小　　结

本章主要介绍了通过链接服务器实现跨实例的连接。使用 SQL Server 中的链接服务器可以链接到 Oracle、DB2、Sybase、Access、Excel 等数据库中，当然也可以链接到另外的 SQL Server 实例中。链接服务器的查询可以直接通过“服务器名”|“数据库名”|“架构名”|“对象名”的形式来查询。为了书写简便和便于以后程序的修改，可以使用同义词来代替。除了使用一般的 SQL 语句访问链接服务器外，SQL Server 提供了 OPENQUERY()函数用于将 SQL 语句发送到链接服务器中，由链接服务器的数据库引擎负责 SQL 查询，然后将查询结果返回给 SQL Server，从而提供了查询的效率。

第 17 章　数据库管理自动化

在数据库的管理和实际应用中，经常需要定期执行数据库备份、数据同步、日志清理、数据内容更改、更新统计信息、重建索引等，这些工作都可以由 SQL Server 代理自动完成。本章将主要讲解数据库管理中的自动化操作。

17.1　SQL Server 代理

SQL Server 代理是一种独立于数据库引擎的 Windows 服务，用于执行安排的管理任务，即“作业”。本节主要介绍 SQL Server 代理的基础知识和如何启用 SQL Server 代理。

17.1.1　SQL Server 代理简介

SQL Server 代理（SQL Server Agent）是一个数据库实例中独立于数据库引擎的 Windows 服务。SQL Server 代理在默认情况下并没有启用，在 SQL Server 代理启用的情况下，它将按照用户定义的计划定期执行代理中定义的作业，其使用 SQL Server 存储作业信息，作业包含一个或多个作业步骤。每个步骤都有自己的任务。

例如，对某个或多个数据库执行备份操作。SQL Server 代理可以按照计划运行作业，也可以在响应特定事件时运行作业，还可以根据需要运行作业。例如，希望在每天晚上 12：00 没有业务处理时，系统能够自动对某个业务数据库执行备份，如果备份出现问题，SQL Server 代理可记录该事件，并通过邮件等方式通知操作员。

SQL Server 代理是存在于实例中的，也就是说，如果一台服务器上安装了多个 SQL Server 实例，则会存在多个 SQL Server 代理。默认实例的 SQL Server 代理在任务管理器中为 SQLAGENT.EXE 进程。

SQL Server 代理使用下列组件定义要执行的任务、执行任务的时间及报告任务成功或失败的方式。

1．作业

“作业”是 SQL Server 代理执行的一系列指定操作。使用作业可以定义一个能执行一次或多次的管理任务，并能监视执行结果成功还是失败。作业可以在一个本地服务器上运行，也可以在多个远程服务器上运行。可以通过下列几种方式运行作业：

- 根据一个或多个计划。
- 响应一个或多个警报。
- 通过执行 sp_start_job 存储过程。

作业中的每个操作都是一个“作业步骤”。例如，作业步骤可以运行 Transact-SQL 脚本、命令行应用程序、Microsoft ActiveX 脚本、Integration Services 包、Analysis Services 命令和查询或复制任务。作业步骤作为作业的一部分进行管理。

2. 计划

“计划”指定了作业运行的时间。多个作业可以根据一个计划运行，多个计划也可以应用到一个作业。计划可以为作业运行的时间定义下列条件：

- 每当 SQL Server 代理启动时。
- 每当计算机的 CPU 使用率处于定义的空闲状态水平时。
- 在特定日期和时间运行一次。
- 按重复执行的计划运行。

3. 警报

“警报”是对特定事件的自动响应。例如，事件可以是启动的作业，也可以是达到特定阈值的系统资源。可以定义警报产生的条件。警报可以响应下列任一条件：

- SQL Server 事件。
- SQL Server 性能条件。
- 运行 SQL Server 代理的计算机上的 WMI 事件。

4. 操作员

操作员定义的是负责维护一个或多个 SQL Server 实例的个人联系信息。在有些企业中，操作员职责被分配给一个人。在拥有多个服务器的企业中，操作员职责可以由多人分担。操作员既不包含安全信息，也不会定义安全主体。

SQL Server 可以通过下列一种或多种方式通知操作员有警报出现：电子邮件、寻呼程序（通过电子邮件）和 Net send。

17.1.2　启用 SQL Server 代理

在 SQL Server 2012 中，出于安全的考虑，默认情况下 SQL Server 代理是被禁用的。通过 SQL Server 配置管理器可以配置和管理 SQL Server 代理。由于 SQL Server 代理本身是一个 Windows 服务，所以通过 Windows 自带的服务管理器也可以管理 SQL Server 代理。使用 SQL Server 配置管理器设置和管理 SQL Server 代理的操作如下。

（1）打开 SQL Server 配置管理器，在 SQL Server 服务界面可以看到当前服务器的 SQL Server 代理服务和运行状态，如图 17.1 所示。

说明： 如果 SQL Server 代理是默认实例，则作为名为 SQL Server 代理（MSSQLSERVER）的服务运行；如果它是命名实例，则作为名为 SQLAgent$<实例名>的服务运行。

（2）选择“SQL Server 代理”节点，然后选择弹出的快捷菜单中的“属性”选项，系统将打开 SQL Server 代理属性对话框，如图 17.2 所示。

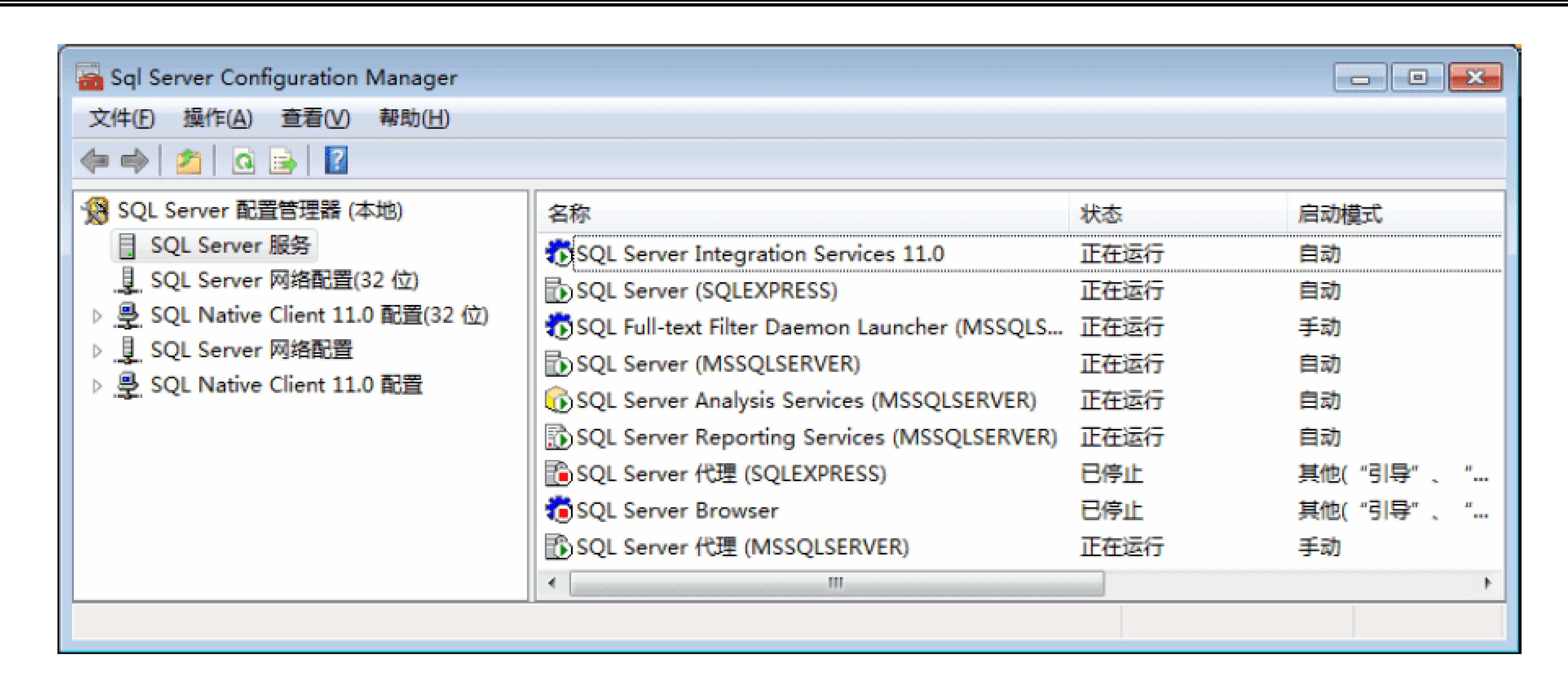


图 17.1　SQL Server 配置管理器中的 SQL Server 代理

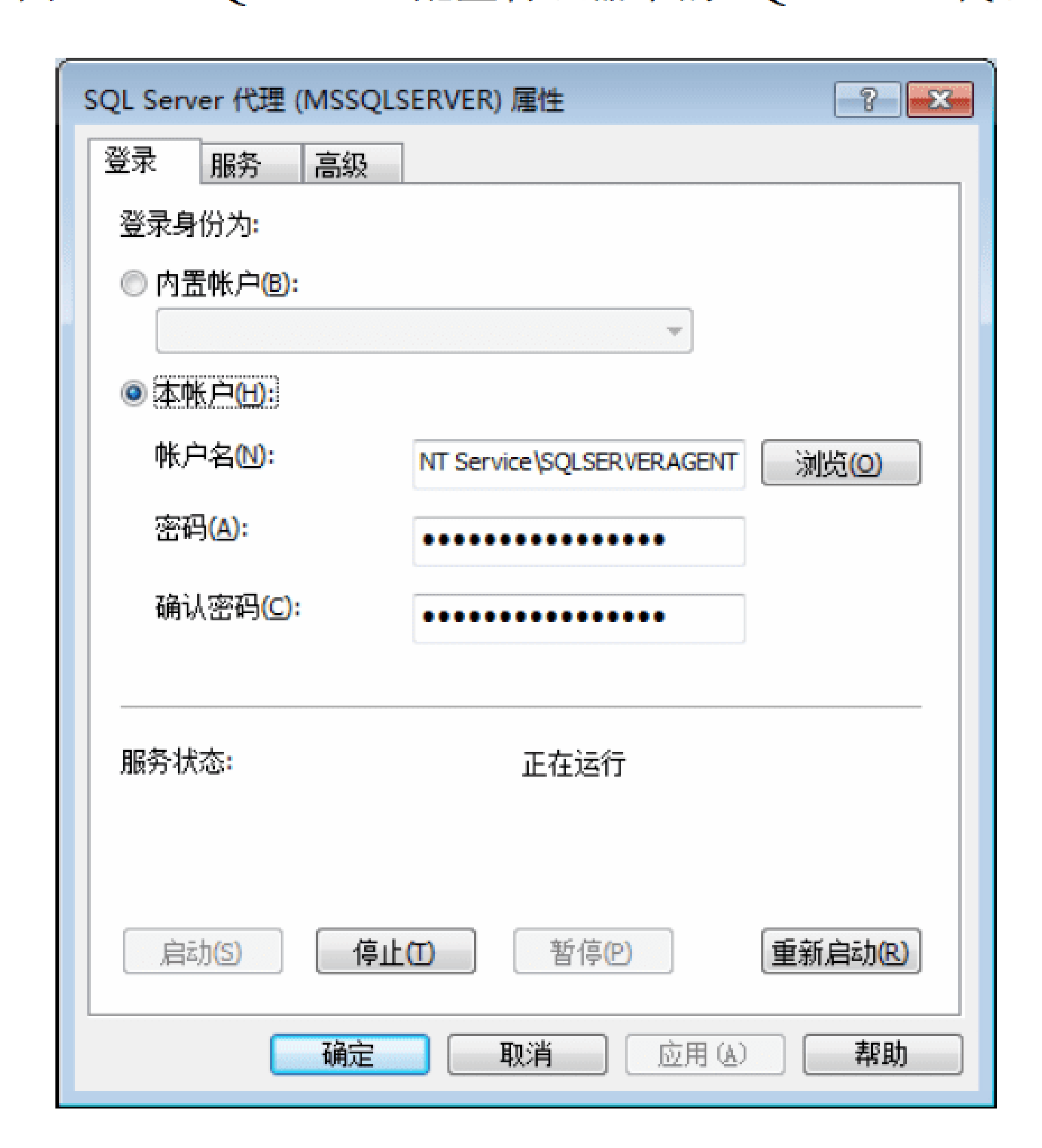


图 17.2　SQL Server 代理属性对话框

（3）在登录界面可以设置运行 SQL Server 代理服务的账户，还可以启动、暂停、停止和重启服务。其中内置账户有如下 3 个账户。

- “本地系统”账户。此账户的名称是 NTAUTHORITY\System。该账户功能强大，可以不受限制地访问所有本地系统资源。它是本地计算机上 Windows Administrators 组的成员，因此也是 SQL Server sysadmin 固定服务器角色的成员。
- “网络服务”账户。此账户的名称是 NTAUTHORITY\NetworkService，可以在 Windows XP 和 Windows Server 2003 中使用。所有使用网络服务账户运行的服务都会验证到作为本地计算机的网络资源。
- “本地服务”账户。此账户的名称是 NTAUTHORITY\LocalService，它作为没有凭据的空会话访问网络资源。

注意：为了避免 SQL Service 代理服务受到资源访问的限制，造成其无法正常运行，微软建议不要对 SQL Server 代理服务使用网络服务账户，更不要为 SQL Server 代理服务配置在本地服务账户下运行。

除了内置账户外，SQL Service 还提供了“本账户”选项。用户可以指定运行 SQL Server 代理服务的 Windows 域账户。如果选择非 Windows Administrators 组成员的 Windows 用户账户，当 SQL Server 代理服务账户不是本地 Administrators 组的成员时，在使用多服务器管理时存在限制。

（4）设置好 SQL Server 代理服务所在的账户后选择“服务”选项卡，进入服务配置界面，如图 17.3 所示。

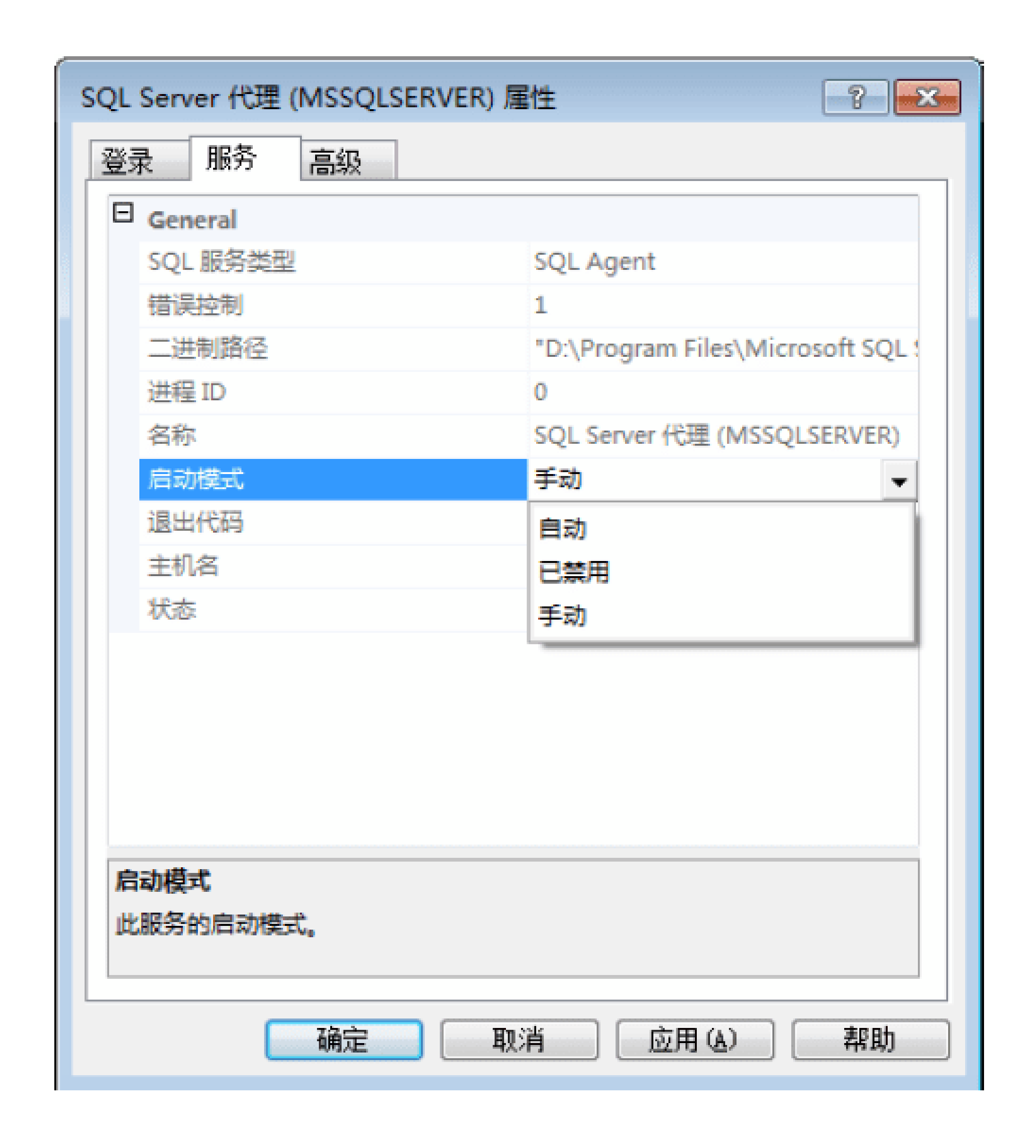


图 17.3　SQL Service 代理服务配置界面

（5）在服务配置界面，只有“启动模式”选项可以修改，其他选项都是只读的。若需要在系统启动的同时启动 SQL Server 代理，选择“自动”选项，否则，若需要手动启动则选择“手动”选项。

（6）“高级”选项卡中主要是配置是否启用错误报告等，与 SQL Server 代理正常使用无关。单击“确定”按钮完成 SQL Server 代理配置。

在配置好 SQL Server 代理后，除了使用 SQL Server 配置管理器和 Windows 的服务管理器外，还可以通过 SSMS 来启动、停止和重启 SQL Server 代理服务。若需要使用 SSMS 管理 SQL Server 代理服务，则必须使用 Windows 身份认证的具有服务管理权限的用户登录。

17.2　配置数据库作业

作业是一系列由 SQL Server 代理按顺序执行的指定操作。作业可以执行一系列活动，包括运行 T-SQL 脚本、命令行应用程序、Microsoft ActiveX 脚本、Integration Services 包、Analysis Services 命令和查询或复制任务。作业可以运行重复任务或那些可计划的任务，它们可以通过生成警报自动通知用户作业状态，从而极大地简化了 SQL Server 管理。

17.2.1　创建作业

SMO、T-SQL 和 SSMS 都可以用于创建作业。使用 T-SQL 创建数据库作业的步骤如下。

（1）执行 sp_add_job 创建作业。

（2）执行 sp_add_jobstep 创建一个或多个作业。

（3）执行 sp_add_schedule 创建计划。

（4）执行 sp_attach_schedule 将计划附加到作业。

（5）执行 sp_add_jobserver 设置作业的服务器。

由于这些存储过程参数众多，设置起来比较复杂，同时 SSMS 提供了将可视化创建的作业编写为 T-SQL 的功能，所以这里就不讲解每一步的操作和每个存储过程的使用方法。读者若需要详细了解可以参考联机丛书。

假设现在有一个电子商务网站，该网站提供了代金券使用的功能，每个代金券都有一个生效和失效日期，用户只能在这期间使用该代金券，未到达生效日期或者已经超过失效日期的代金券其状态将变为不可用。对应的数据库中代金券表的架构如代码 17.1 所示。

代码 17.1　示例代金券表架构

```
CREATE TABLE Coupon
(
    Number char(20) PRIMARY KEY,
    FaceValue money NOT NULL,
    BeginDate datetime NOT NULL,
    EndDate datetime NOT NULL,
    Status tinyint NOT NULL
)
```

这种情况可以在用户使用每一张代金券的时候都去判断生效日期、失效日期和当前时间的关系，然后决定是否可用。但是这种方法效率低下，增加了程序逻辑的复杂度。

较好的一种解决方案是在 SQL Server 中创建一个数据库作业，该数据库作业负责将未生效和已经失效的代金券状态修改为不可用，将到达生效日期并且没有失效的代金券状态修改为可用。这样程序只需要判断代金券的状态是否可用，而不需要进行复杂的日期判断。使用 SSMS 创建作业的操作步骤如下。

（1）在 SSMS 的对象资源管理器中展开“SQL Server 作业”节点。

（2）选择其中的“作业”子节点，在弹出的快捷菜单中选择“新建作业”选项，系统将弹出“新建作业”对话框，如图 17.4 所示。

（3）在“名称”文本框中输入作业的名称，如这里要修改代金券的状态，则可以将作业名称命名为 ChangeStatus，当然也可以使用中文。

（4）“所有者”文本框中指定了拥有该作业的用户，默认情况下就是当前登录用户，不需要修改。

（5）“类别”下拉列表框中列出了作业的所有分类，该分类并不会对作业的运行有任何影响，只是便于作业的分类管理。这里保持默认值未分类。

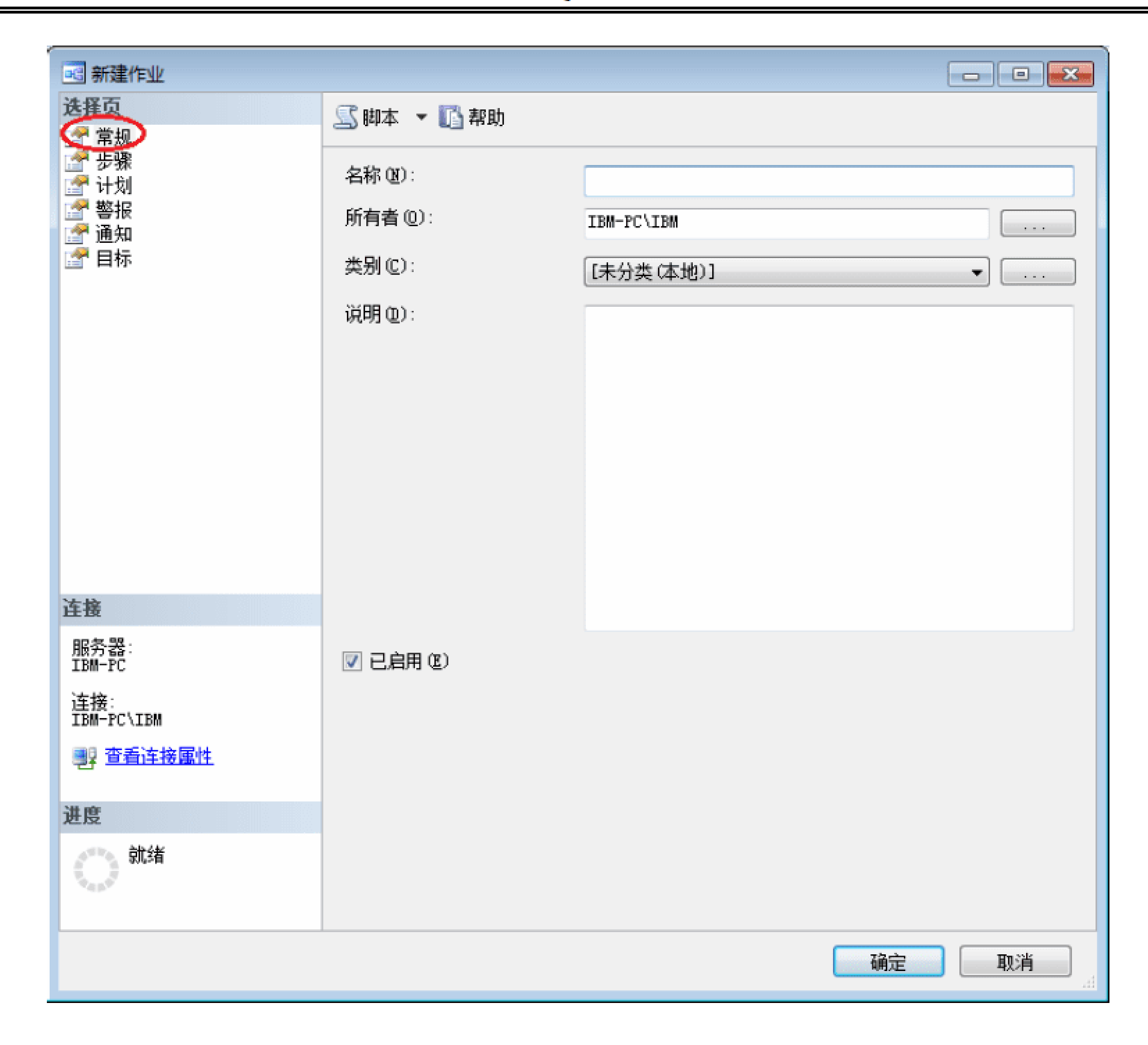


图 17.4 “新建作业”对话框

（6）如果希望创建后的作业并不运行，则取消选中“已启用”复选框。

（7）“说明”文本框中可以输入对该作业的描述等说明性文字，最多 512 字。

17.2.2　创建作业步骤

数据库作业最基本的要素就是步骤，作业步骤是作业对数据库或服务器执行的操作，每个作业必须至少有一个作业步骤。作业步骤可以为：

- ❑ 可执行程序和操作系统命令（CmdExec）。
- ❑ T-SQL 语句，包括存储过程和扩展存储过程。
- ❑ PowerShell 脚本。
- ❑ Microsoft ActiveX 脚本。
- ❑ 复制任务。
- ❑ Analysis Services 任务。
- ❑ Integration Services 包。

在 SSMS 创建更改代金券状态作业步骤的操作如下。

（1）在“新建作业”窗口中选择“步骤”选项，系统切换到步骤设置界面，如图 17.5 所示。

（2）单击“新建”按钮，进入“新建作业步骤”对话框，如图 17.6 所示。

（3）在“步骤名称”文本框中输入要创建的步骤的名字，比如 ChangeStatusActive，用以说明该步骤用于将代金券状态改为已激活。在“类型”下拉列表框中选择“Transact-SQL 脚本（T-SQL）”选项。

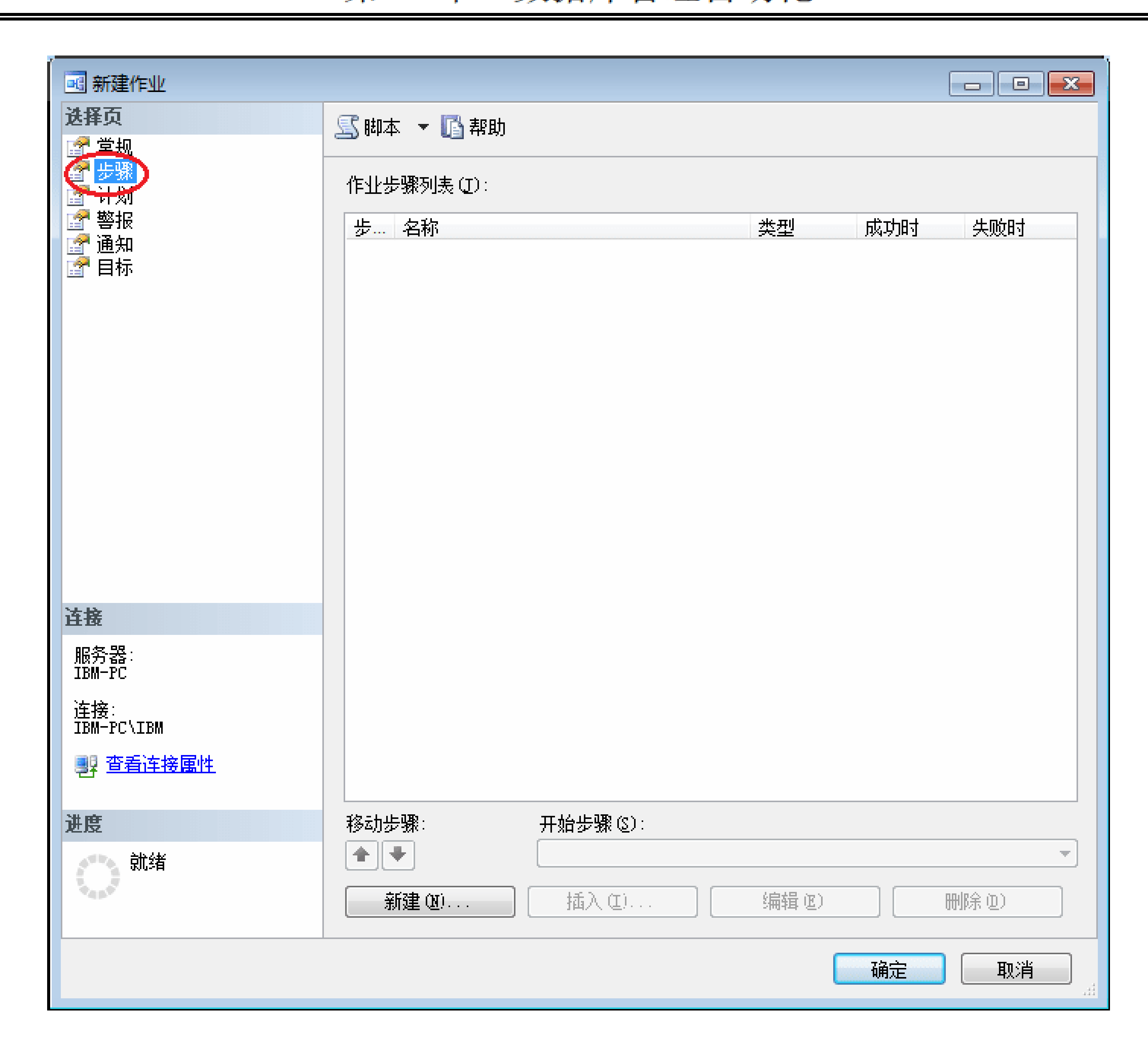


图 17.5　作业步骤设置界面

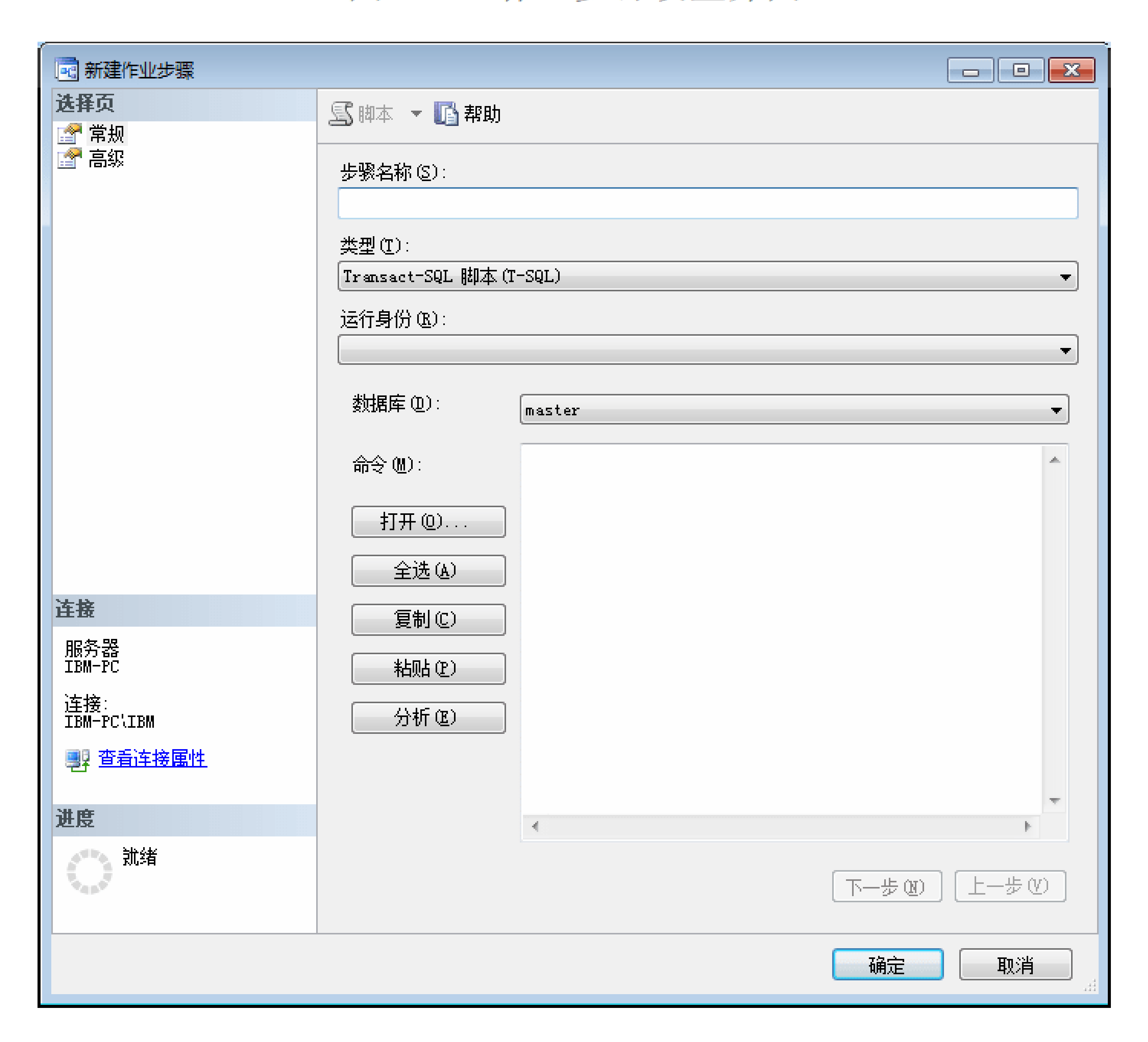


图 17.6　“新建作业步骤”对话框

（4）在“数据库”下拉列表框中选择需要运行该 T-SQL 语句的数据库。在“命令”文本框中输入要运行的 SQL 语句，将数据库中代金券的状态由未激活改为已激活的 SQL，脚本如代码 17.2 所示。

代码 17.2　激活代金券的脚本

```
DECLARE @now DATETIME
SET @now = GETDATE()
UPDATE dbo.Coupon
```

```
SET  Status = 1 --状态修改为已激活
WHERE  Status = 0 --原来的状态是未激活的
 AND  @now BETWEEN BeginDate AND EndDate
```

（5）选择“高级”选项页，进入作业步骤高级配置界面，如图 17.7 所示。

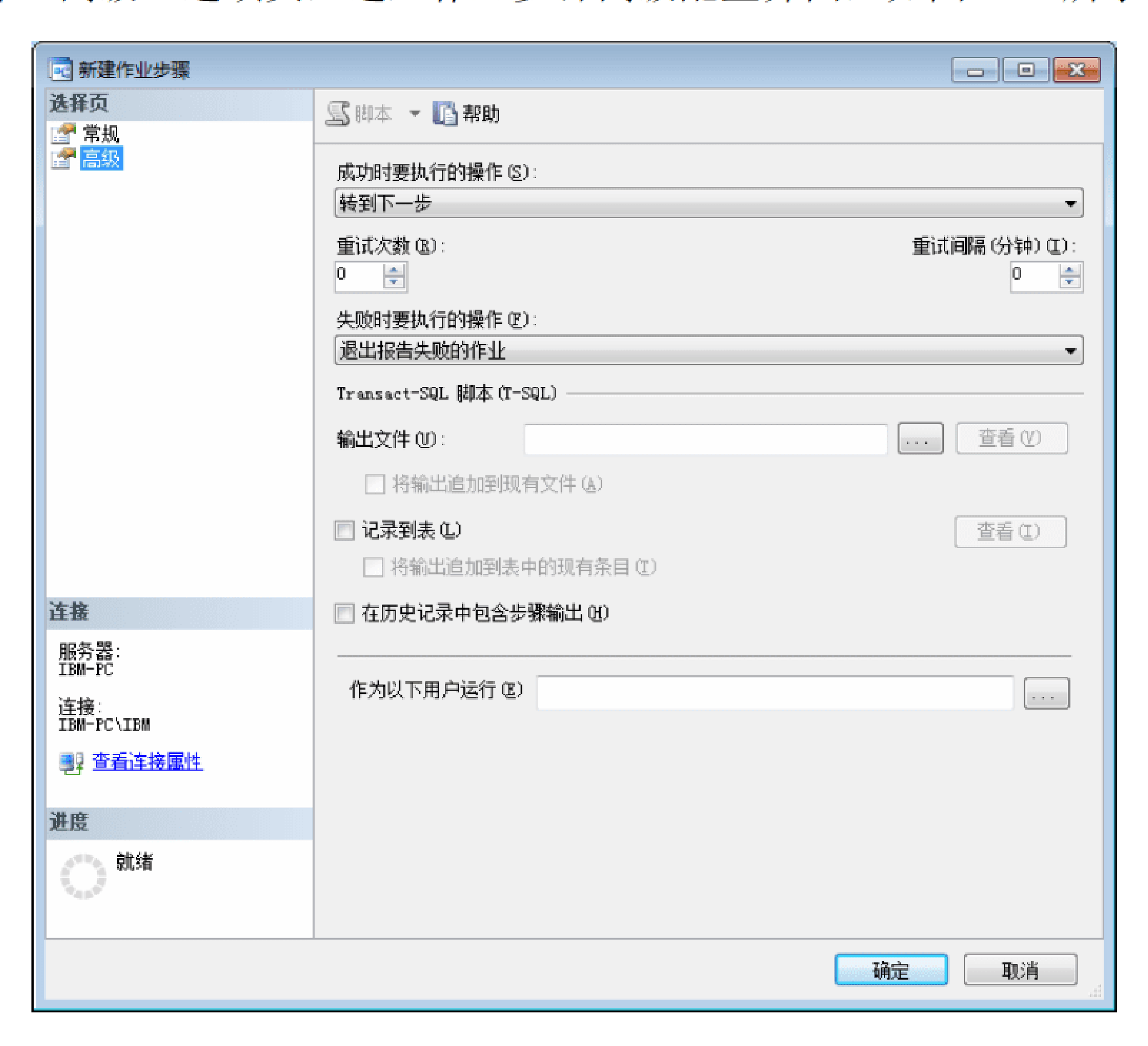


图 17.7　作业步骤高级配置界面

（6）在“成功时要执行的操作”下拉列表框中提供了“转到下一步”、“退出报告成功的作业”和“退出报告错误的作业”3 个选项。对于多个步骤的作业，除了“转到下一步”外还可以指定转到另外的步骤。如果当前步骤不是最后一步，则应该选择“转到下一步”选项，选择其他两个选项都会造成下面的步骤不被运行。由于这里需要创建两个步骤，一个用于激活代金券，另一个用于将过期的代金券设置为失效，所以这里选择“转到下一步”选项。

（7）接下来是配置如果当前步骤执行失败时执行的操作。“重试次数”是指如果执行失败，则重试执行当前步骤的次数，如果在重试次数内一旦成功，则认为该步骤执行成功，否则认为当前步骤执行失败。“重试间隔”是每次重试之间的时间间隔。“失败时要执行的操作”提供的选项与“成功时要执行的操作”的选项相同，如果当前步骤与下一步骤之间没有必然的先后顺序或业务联系，那么可以设置失败时“转到下一步”以继续执行下面的步骤。默认情况下，失败时都是“退出报告失败的作业”。

（8）在“输出文件”文本框中可以设置将当前步骤执行的日志写入指定的文件中。“记录到表”是将作业步骤的输出记录到 msdb 数据库的 sysjobstepslogs 表中。在至少运行了一次作业后，便可单击“记录到表”旁边的“查看”按钮来查看作业步骤的输出，当然也可以直接利用 SELECT 查询 sysjobstepslogs 表。“在历史记录中包含步骤输出”复选框表示在作业历史记录中不仅仅包含运行的日志，还包含作业步骤执行后的输出。“作为以下用户运行”选项可以指定执行当前步骤的数据库用户。

（9）单击“确定”按钮回到图 17.5 所示的界面。再新建一个作业步骤 ChangeStatusDisable 用于将过期的代金券设置为失效，其脚本如代码 17.3 所示。

代码 17.3　使过期代金券失效

```
DECLARE @now DATETIME
SET @now = GETDATE()
UPDATE dbo.Coupon
SET    Status = 2 --状态修改为失效
WHERE  Status = 1 --原来的状态是已激活的
  AND  @now > EndDate
```

由于整个作业只有这两个步骤，所以在该步骤的高级选项页中的“成功时要执行的操作”下拉列表框中选择“退出报告成功的作业”选项。

（10）在“作业步骤列表”列表框下可以指定作业开始步骤，默认为创建的第一个步骤，如图 17.8 所示。

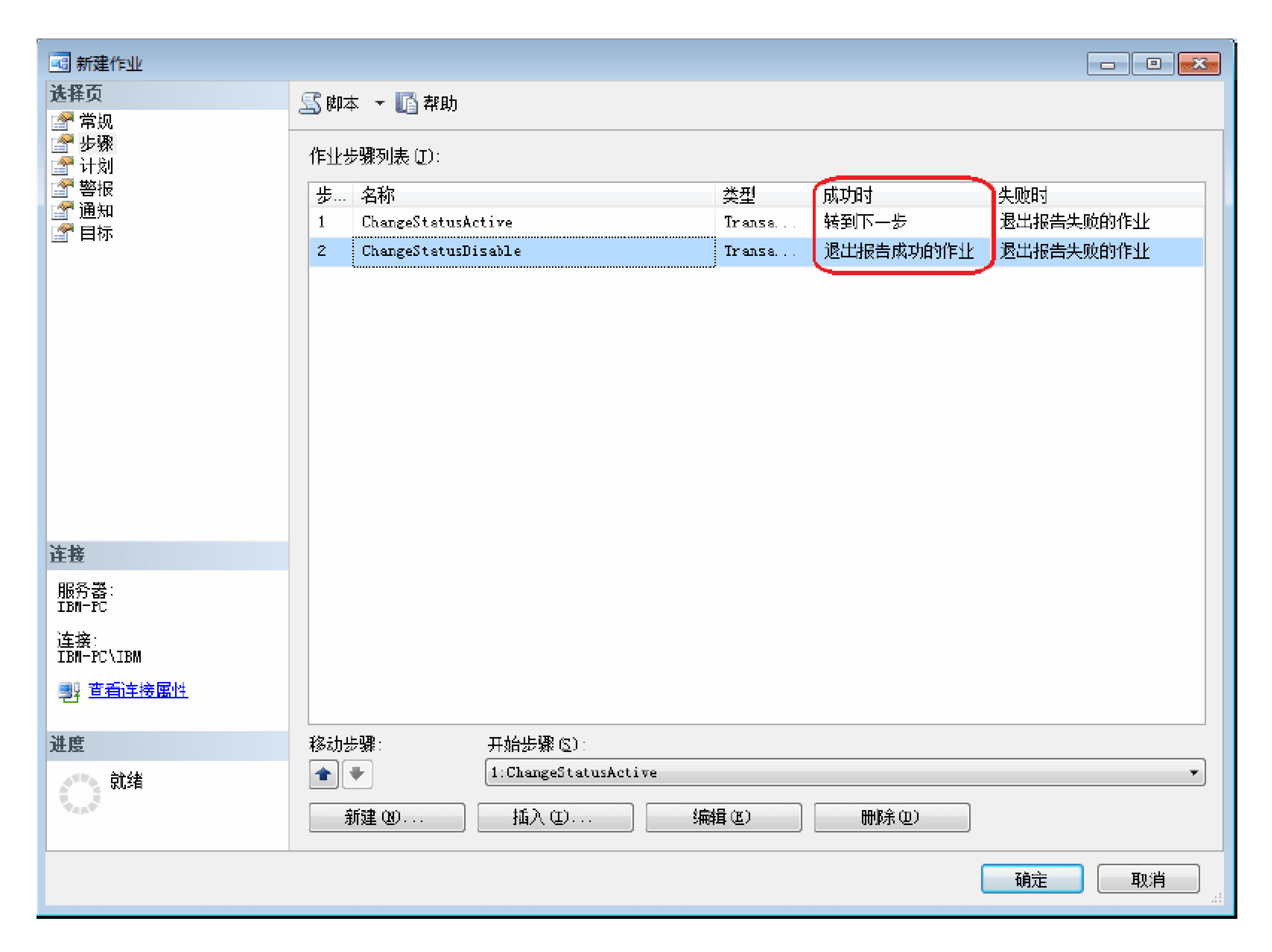


图 17.8　作业步骤列表

（11）单击“确定”按钮系统将创建包含这两个作业步骤的数据库作业 ChangeStatus，同时在 SSMS 的对象资源管理器中可以看到该作业出现在“作业”节点下。

17.2.3　创建计划

计划是作业运行的时间。在前面创建的作业中并没有创建计划，所以该作业在没有人为干预的情况下将永远也不会运行。计划和作业在 SQL Server 代理中是平等的多对多关系，也就是说计划并不依赖于作业而存在，而且一个作业可以关联多个计划，一个计划也可以执行多个作业。

前面创建的作业用于每日定时清理代金券的状态，所以需要创建一个计划，在每天晚上 12 点的时候运行该作业。使用 SSMS 创建计划的操作如下。

（1）在 SSMS 的对象资源管理器中展开“SQL Server 代理”节点。选择其中的“作业”子节点，在弹出的快捷菜单中选择“管理计划”选项，系统将弹出“管理计划”对话框，如图 17.9 所示。

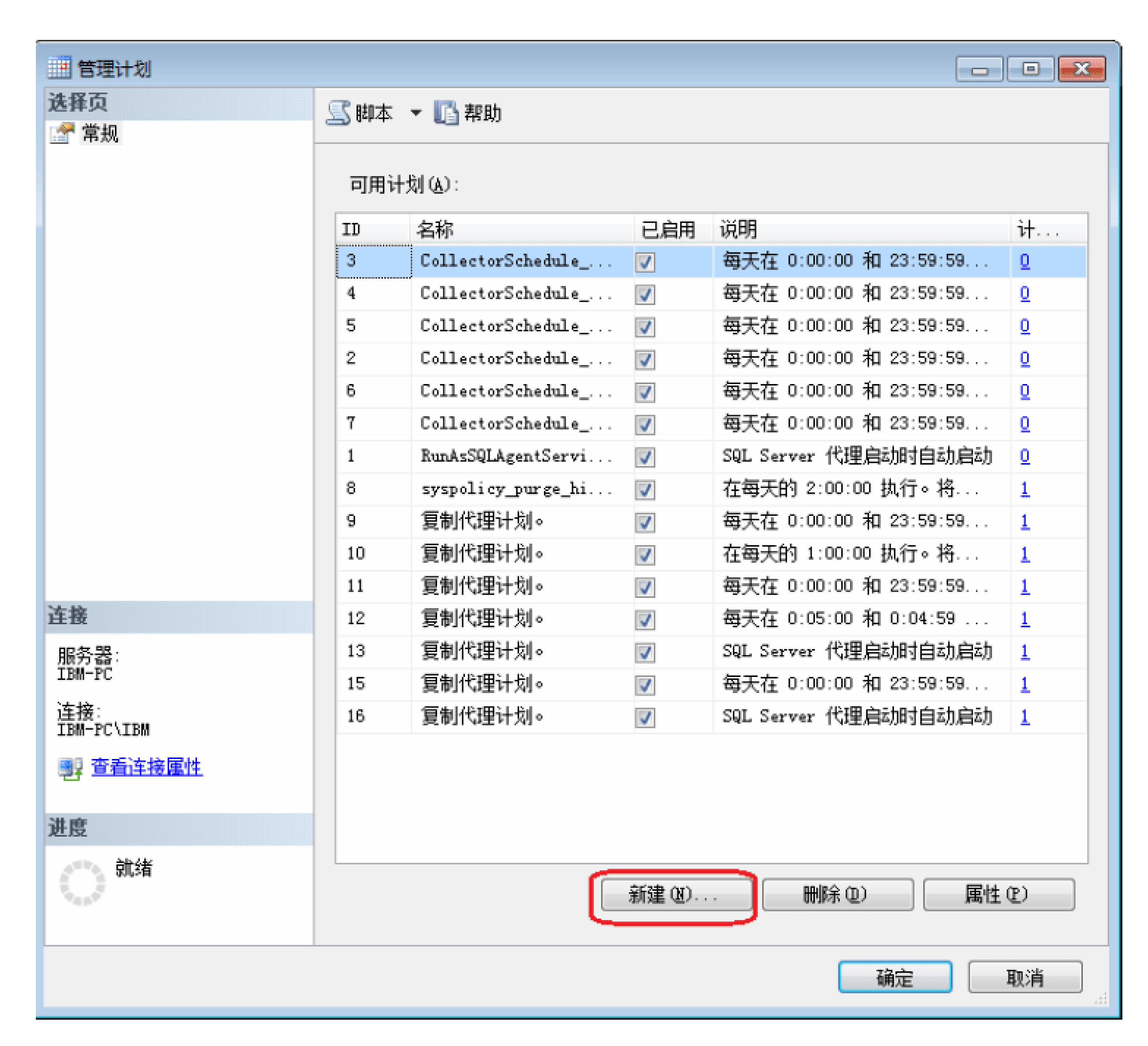


图 17.9　“管理计划”对话框

（2）其中在“可用计划”列表框中列出了当前 SQL Server 代理中的所有计划，如果有现成的每晚 12 点执行的计划，那么就可以复用这个计划了。由于这里没有，所以需要创建新的计划。单击“新建”按钮，进入“新建作业计划”对话框，如图 17.10 所示。

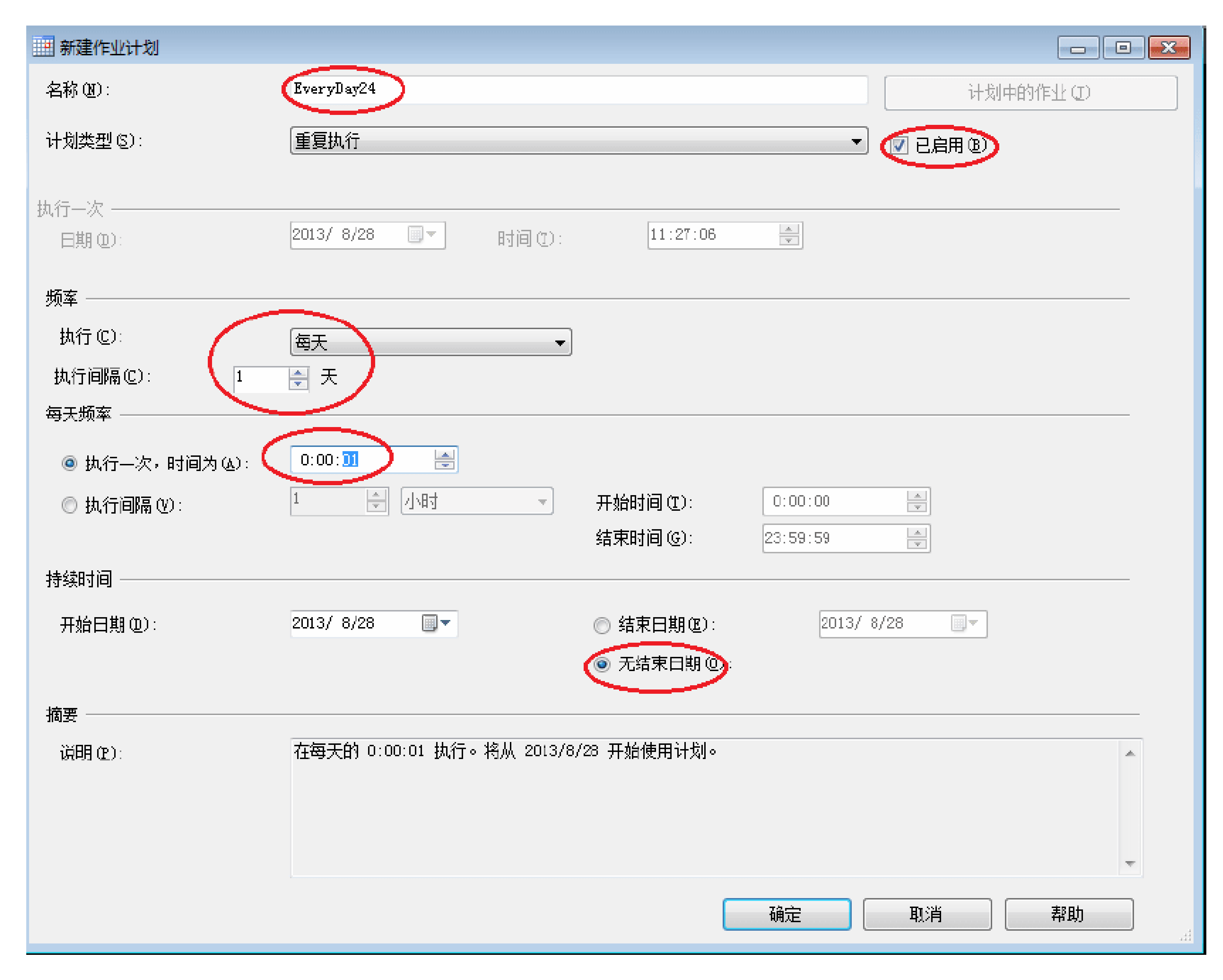


图 17.10　“新建作业计划”对话框

（3）在“名称”文本框中输入该计划的名称，这个名称只要能够简单表达该计划的内容或作用即可。在“计划类型”下拉列表框中提供了 SQL Server 代理启动时自动启动、CPU 空闲时启动、重复执行和执行一次 4 个选项。由于这里的作业每天晚上都要执行，所以选择“重复执行”选项。旁边的“已启用”复选框表示该计划是否在创建后启用。

（4）SQL Server 在执行频率上提供了每天、每周和每月 3 个选项，不同的选项对应不同的子选项。每天执行需要指定执行间隔，如果是隔一天运行一次，则执行间隔为 2 天；如果是每天都运行，则执行间隔为 1 天；如果是每周执行，则子选项如图 17.11 所示。

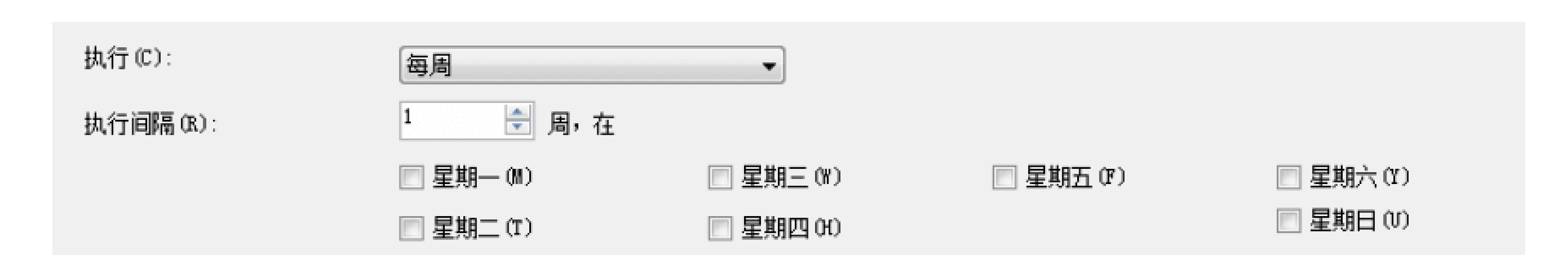


图 17.11　每周执行间隔配置

如果某些处理是在每周末没有业务数据运行的情况下运行，可以选择执行间隔 1 周，在星期六执行（选星期天也可以），这样每个周六的时候该计划将会运行。如果是每月执行，则子选项如图 17.12 所示。



图 17.12　每月执行间隔配置

比如每个月 25 日是结账期，需要在每个月的第 25 天执行作业，则可以选择“天”单选按钮，然后天选择 25，月选择 1。如果作业是每个月的最后一周星期一执行，则可以选择下一个单选按钮。

（5）在“每天频率”选项区域中可以配置到了执行计划的日期，作业在该日期执行的频率。由于更新代金券状态的作业只需要每天运行一次，所以选择执行一次单选按钮，时间为晚上 12 点，也就是 00：00：01。如果是频繁执行，比如每个小时都要执行一次，那么就应该将执行间隔配置为 1 小时，同时还可以配置开始和结束时间。

注意：这里在时间配置上随意性比较大，不管是配置成了晚上 01：00：00 还是 00：30：00 都可以，但如果是多个作业之间相互协调处理，则要注意安排作业执行时的先后顺序，最好不要将所有作业都安排在同一个时间运行。

（6）“持续时间”选项区域用于配置该计划的开始和结束日期，如果计划需要长期地运行，则选择“无结束日期”单选按钮。

（7）单击“确定”按钮，新建作业计划完成，当前计划将出现在计划列表中，如图 17.13 所示。

（8）计划列表中刚创建的 EveryDay24 计划对应的“计划中的作业”为 0，表示当前计划并没有关联任何作业。单击“0”链接，弹出引用计划的作业对话框，如图 17.14 所示。

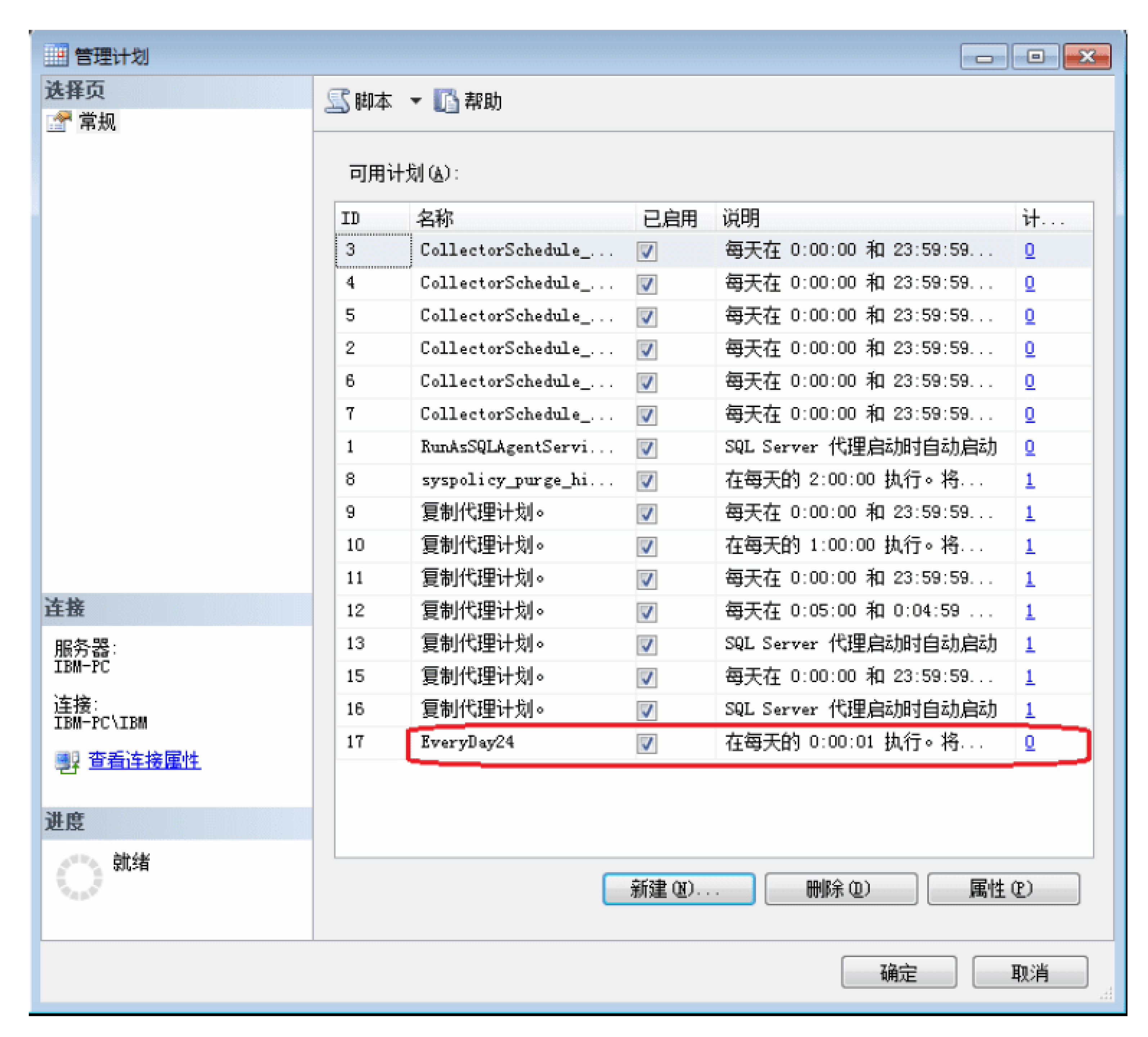


图 17.13　创建成功的作业

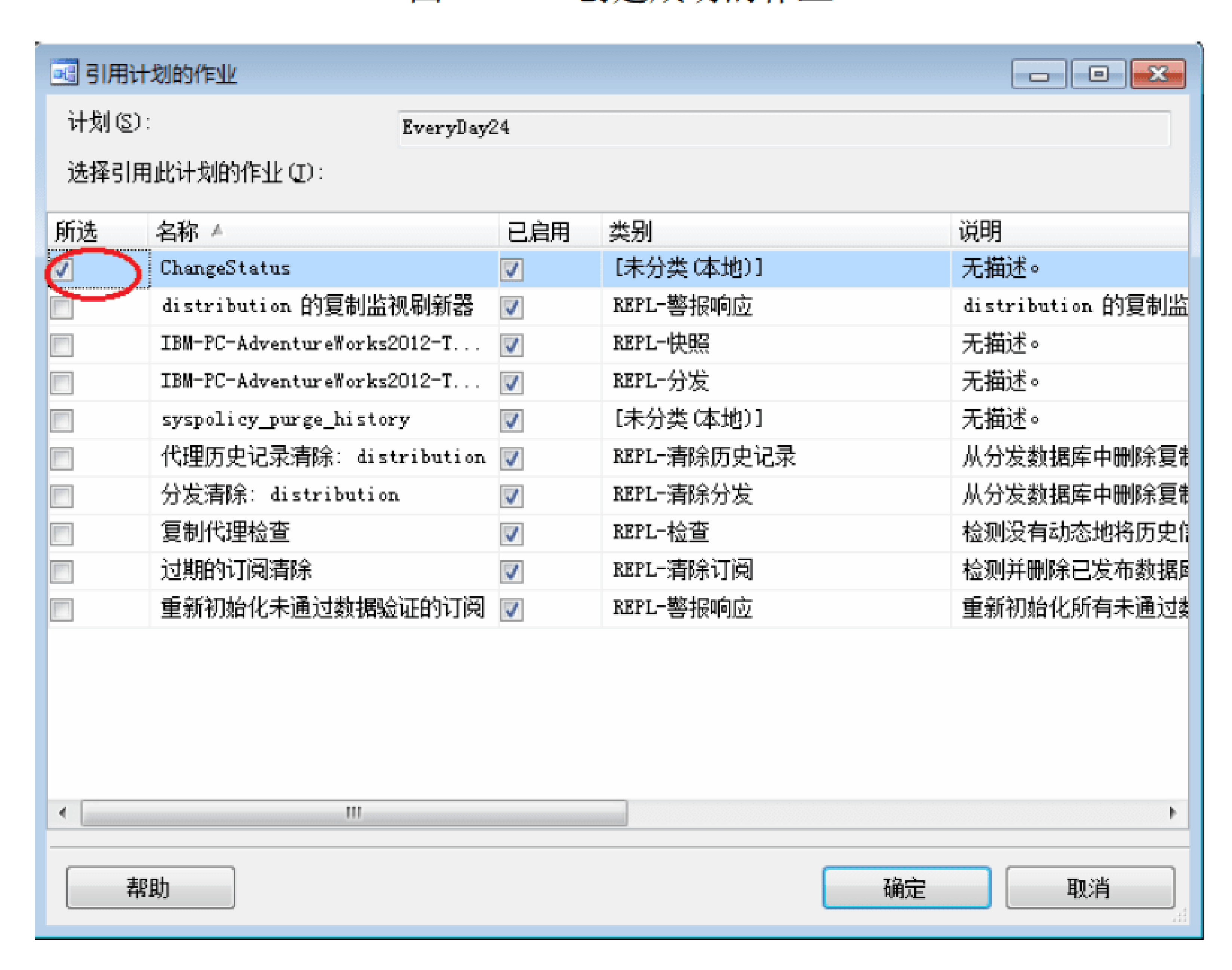


图 17.14　“引用计划的作业”对话框

（9）选中要关联的作业 ChangeStatus 左边的复选框，然后单击“确定”按钮，便可将作业与计划关联，回到“管理计划”对话框。

（10）单击“管理计划”对话框中的“确定”按钮，完成整个计划的创建和关联作业的操作。

除了通过“管理计划”对话框来设置作业和计划的关联外，还可以使用作业属性对话框中的“计划”选项页来创建和管理计划，如图 17.15 所示。

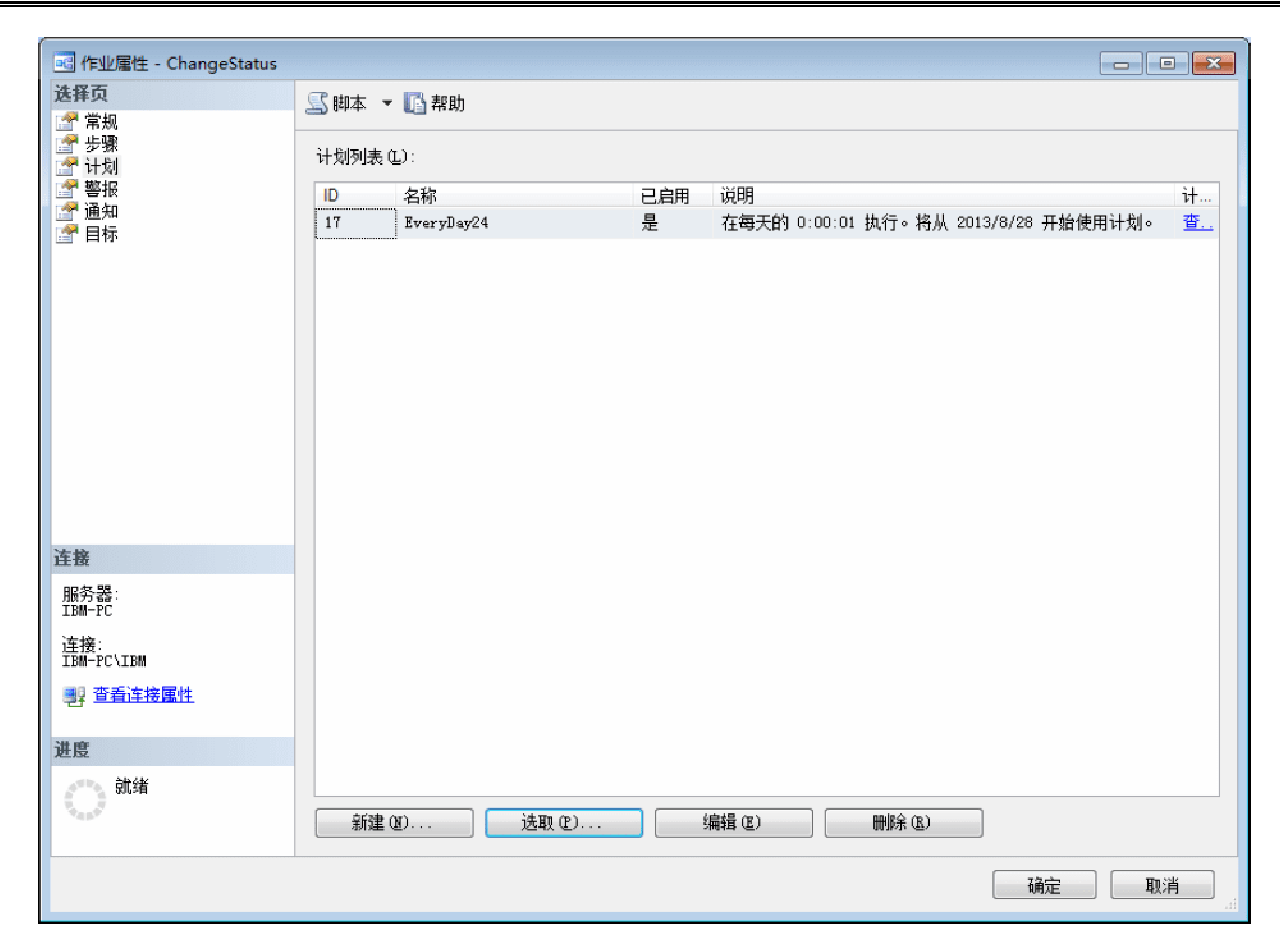


图 17.15　作业属性对话框的“计划”选项页

其中的“新建”、“编辑”和“删除”按钮分别用于执行对计划的增加、修改和删除操作，而“选取”按钮用于选择关联已有的计划。

17.2.4　运行作业

作业在指定关联计划后便会按照计划中的配置定时运行，但是有时需要人为地运行作业。如果是使用应用程序来运行作业，则需要执行系统存储过程 sp_start_job。该存储过程在 msdb 数据库，其语法格式如代码 17.4 所示。

代码 17.4　sp_start_job 语法格式

```
sp_start_job
    {   [@job_name =] 'job_name'
      | [@job_id =] job_id }
    [ , [@error_flag =] error_flag]
    [ , [@server_name =] 'server_name']
    [ , [@step_name =] 'step_name']
    [ , [@output_flag =] output_flag]
```

其中最主要的参数@job_name 是要执行的作业名称。@server_name 可以用于指定启动作业的目标服务器。@step_name 用于指定开始执行作业的步骤名。

例如要运行前面创建的作业 ChangeStatus，则对应执行的脚本为：

```
EXEC msdb.dbo.sp_start_job 'ChangeStatus'
```

注意：手动启用作业将以异步的方式运行，也就是说作业一旦被成功启动，该存储过程就完成了，而接下来作业是否能够成功执行完所有作业步骤，该存储过程将无法得知，可以使用 sp_help_job 查询作业的信息。

若作业一直处于运行状态或者被阻塞，需要将作业停止，则可以使用系统存储过程 sp_stop_job，该存储过程的使用语法如代码 17.5 所示。

代码 17.5　sp_stop_job 语法

```
sp_stop_job
     [@job_name =] 'job_name'
   | [@job_id =] job_id
   | [@originating_server =] 'master_server'
   | [@server_name =] 'target_server'
```

其中最重要的参数就是@job_name，例如在启用作业 ChangeStatus 后停止该作业的脚本为：

```
EXEC msdb.dbo.sp_stop_job 'ChangeStatus'
```

如果当前作业并没有在运行中，使用该存储过程试图停止作业将会抛出异常。

前面创建的数据库作业在默认情况下是启用的，如果不希望某作业运行，则可以禁用该作业。禁用数据库作业使用系统存储过程 sp_update_job。该存储过程除了用于启用或禁用作业的状态外，还可以更改作业的名字、描述、开始步骤、分类等作业的属性。sp_update_job 的语法如代码 17.6 所示。

代码 17.6　sp_update_job 语法

```
sp_update_job [ @job_id =] job_id | [@job_name =] 'job_name'
     [, [@new_name =] 'new_name' ]
     [, [@enabled =] enabled ]
     [, [@description =] 'description' ]
     [, [@start_step_id =] step_id ]
     [, [@category_name =] 'category' ]
     [, [@owner_login_name =] 'login' ]
     [, [@notify_level_eventlog =] eventlog_level ]
     [, [@notify_level_email =] email_level ]
     [, [@notify_level_netsend =] netsend_level ]
     [, [@notify_level_page =] page_level ]
     [, [@notify_email_operator_name =] 'email_name' ]
         [, [@notify_netsend_operator_name =] 'netsend_operator' ]
         [, [@notify_page_operator_name =] 'page_operator' ]
     [, [@delete_level =] delete_level ]
     [, [@automatic_post =] automatic_post ]
```

只需要将作业的 ID 或者名字和需要修改的新作业属性作为参数传入即可，例如要禁用 ChangeStatus 作业，对应的脚本为：

```
EXEC msdb.dbo.sp_update_job @job_name='ChangeStatus',@enabled=0
```

注意：禁用该作业后作业将不会按计划中的时间来运行，但是用户仍然可以通过手动的方式运行该作业。

若使用 sp_update_job 修改作业的描述，则对应的脚本为：

```
EXEC msdb.dbo.sp_update_job @job_name='ChangeStatus',
@description=N'更新代金券状态的作业'
```

要重新启用作业，则只需要将@enabled 参数设置为 1 即可。

通过 SSMS 来运行、停止、禁用和启用作业的操作将简单许多。若要运行 ChangeStatus 作业，则只需要在对象资源管理器中右击 ChangeStatus 作业，在弹出的快捷菜单中选择“作

业开始步骤”选项，系统将弹出开始作业对话框，其中显示了选择开始执行的步骤，如图 17.16 所示。

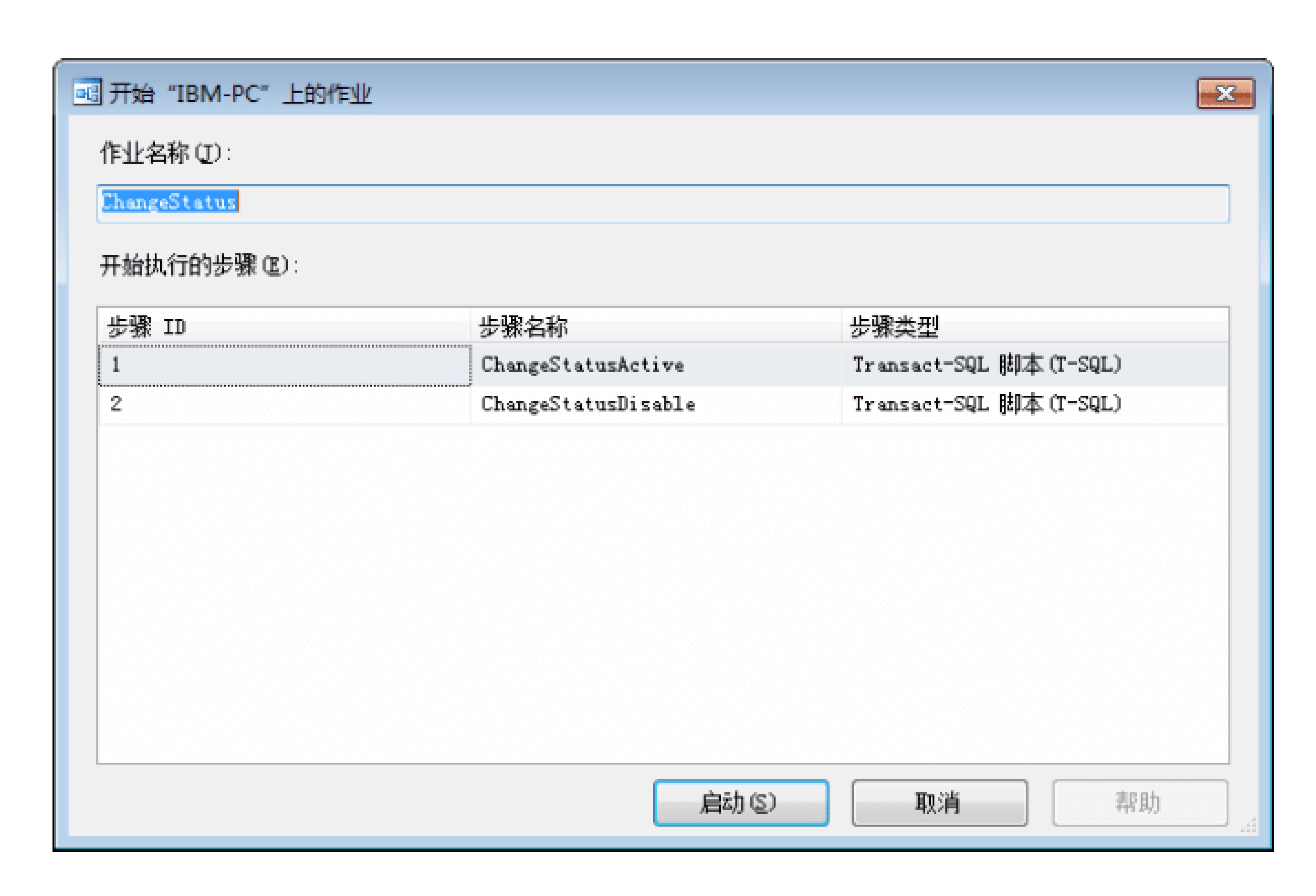


图 17.16　开始作业对话框

选择第 1 个步骤作为作业的开始步骤，然后单击“启动”按钮，系统将开始运行作业并在成功运行作业后返回成功信息。停止、禁用和重新启用作业，都可以通过右击具体的作业，在弹出的快捷菜单中选择相应选项来完成。另外，是否启用作业也可以在作业的属性对话框中修改。

17.2.5　监视作业

每当服务启动时，SQL Server 代理都会创建新的会话。创建新会话后，将用所有存在的已定义的作业填充 msdb 数据库中的 sysjobactivity 表。当 SQL Server 代理重新启动时，此表将保留作业的上一次活动。每个会话均记录从作业开始到作业结束的 SQL Server 代理的正常作业活动。

使用作业活动监视器可以查看 sysjobactivity 表。通过作业活动监视器可以查看服务器上的所有作业，或定义筛选器以限制显示的作业数。还可以通过选择“代理作业活动”网格中的某个列标题对作业信息进行排序。使用作业活动监视器，可以执行下列任务：

- 启动和停止作业。
- 查看作业属性。
- 查看特定作业的历史。
- 手动刷新“代理作业活动”网格中的信息，或者通过选择“查看刷新设置”命令设置自动刷新间隔。

若要查看计划运行的作业、当前会话期间运行的作业的最新结果，以及当前正在运行或空闲的作业，都可以使用作业活动监视器。

在 SSMS 的对象资源管理器中选择“SQL Server 代理”|“作业活动监视器”节点便可打开“作业活动监视器”对话框，如图 17.17 所示。

作业活动监视器中列出了当前代理的所有作业和作业的状态、上次运行结果和运行时间、下次运行时间、类别、是否可以运行、是否关联计划等信息。在作业活动监视器中右击某个

作业，在弹出的快捷菜单中选择相应选项来开始作业步骤、停止作业、启用与禁用作业。

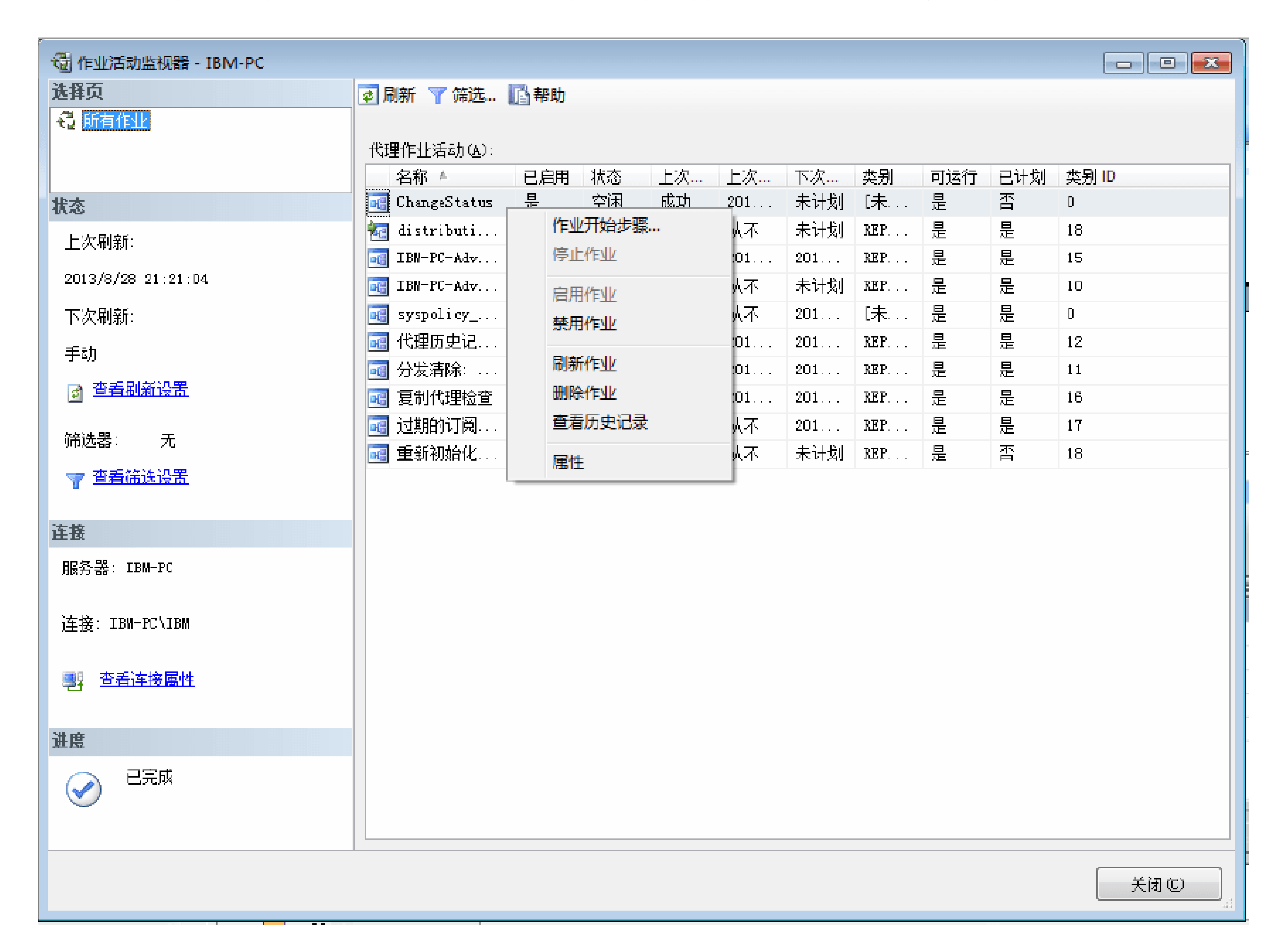


图 17.17　“作业活动监视器”对话框

在弹出的快捷菜单中选择“查看历史记录”选项，系统将弹出“日志文件查看器”对话框，如图 17.18 所示。

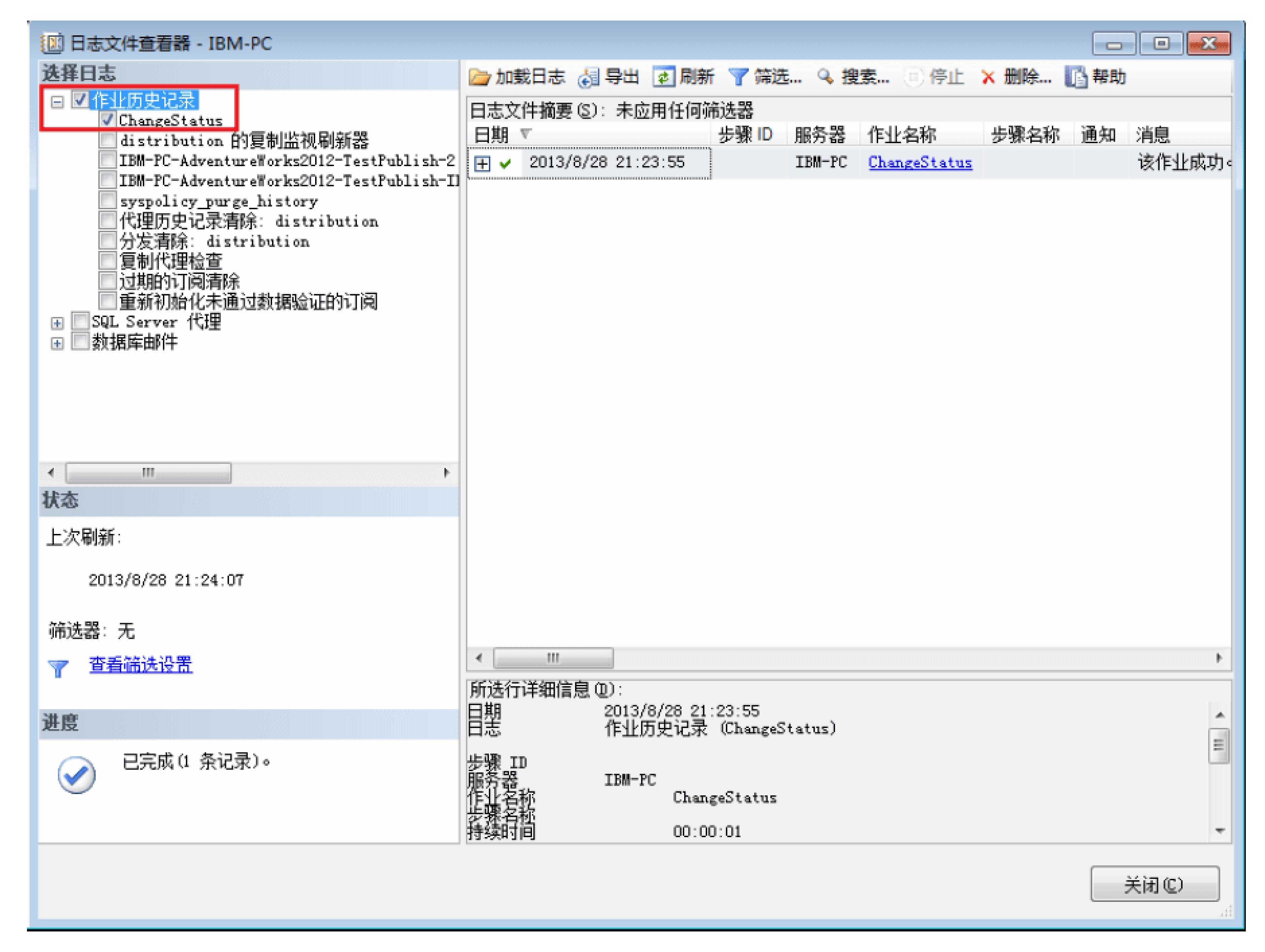


图 17.18　查看作业历史记录

通过选中左上角的作业名可以查看其他作业的历史记录，日志文件摘要列表中默认情况下将列出选中作业的所有历史记录，按照时间倒序排列。如果历史记录过多不便于查看，可以单击“筛选”按钮设置显示的筛选规则。

展开日志文件摘要可以看到作业下每个步骤的执行情况，下面则显示了作业步骤的详细信息。

技巧：要查看作业历史记录，不需要通过作业活动监视器来打开记录，只需要在对象资源管理器中右击具体的作业，然后在弹出的快捷菜单选择的“查看历史记录”选项即可。

17.3　数据库邮件

数据库邮件是从 SQL Server 数据库引擎中发送电子邮件的企业解决方案。通过使用数据库邮件，数据库应用程序可以向用户发送电子邮件。邮件中可以包含查询结果，还可以包含来自网络中任何资源的文件。

17.3.1　数据库邮件简介

在 SQL Server 2005 之前在数据库中发送邮件主要通过 SQL Mail 来完成，SQL Mail 使用外部电子邮件应用程序中的扩展 MAPI 客户端组件，来发送和接收电子邮件。因此，若要使用 SQL Mail，必须在运行 SQL Server 的计算机上安装支持扩展 MAPI 的电子邮件应用程序。

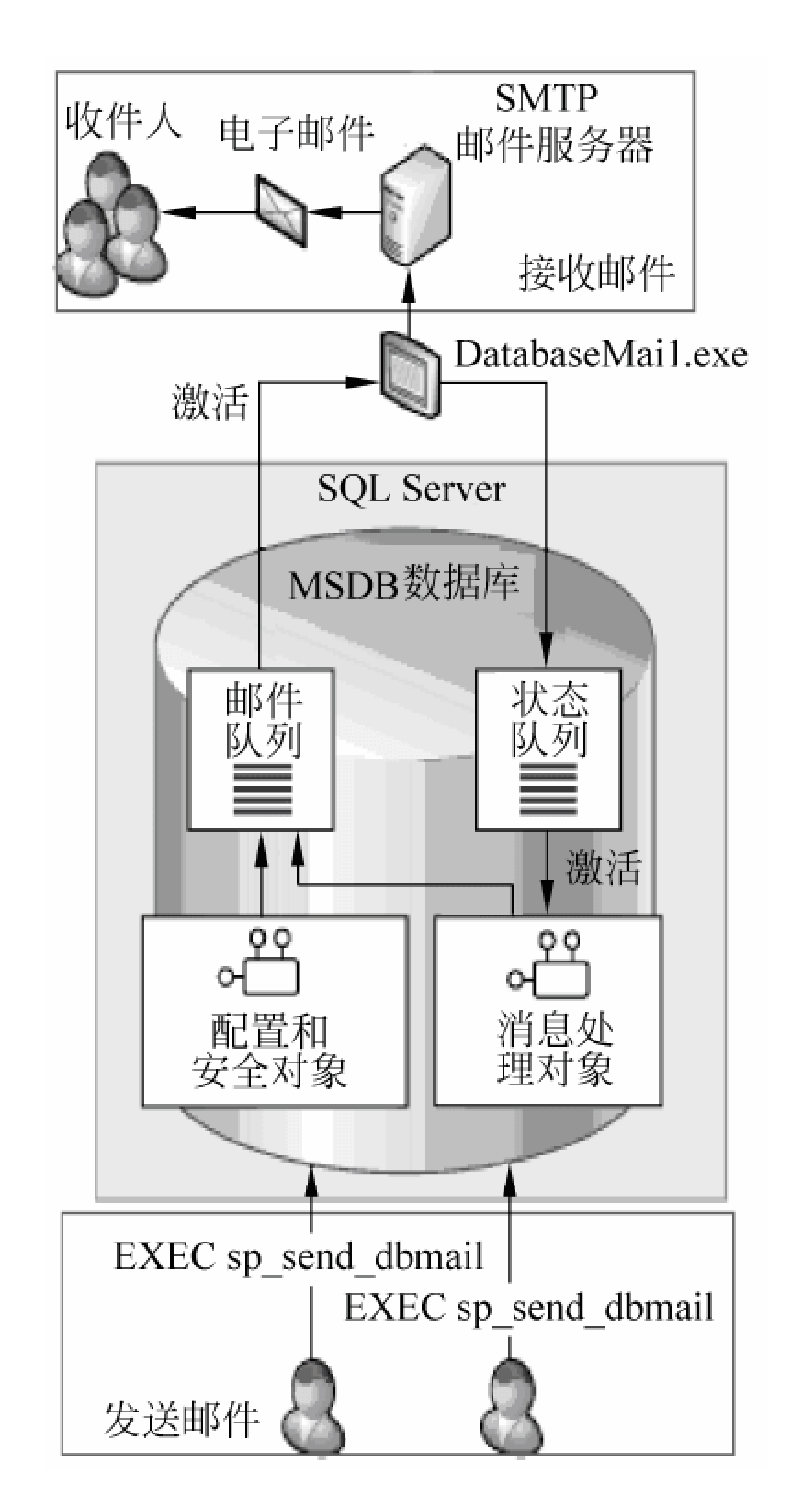
为了保存向后的兼容性，在 SQL Server 2008 中仍然提供了 SQL Mail，但是在 SQL Server 2012 中就已经废除该功能了，而是使用数据库邮件来代替 SQL Mail。使用数据库邮件具有以下优点：

- 数据库邮件使用标准的简单邮件传输协议（SMTP）发送邮件，无须 Microsoft Outlook 或扩展消息处理应用程序编程接口。
- 数据库邮件的进程与 SQL Server 数据库引擎的进程是隔离的。为了尽量减少对 SQL Server 的影响，电子邮件的发送是由单独的程序来完成的。即使该程序停止、失败或者发生异常，SQL Server 也会继续将电子邮件放入队列，下次再尝试发送。
- 数据库邮件配置文件允许指定多台 SMTP 服务器，如果一台 SMTP 服务器不可用，还可以使用其他的 SMTP 服务器发送。
- 群集支持。数据库邮件与群集兼容，并且可以完全用于群集中。
- 支持多个数据库邮件配置文件。在一个 SQL Server 实例中可以创建多个数据库邮件配置文件，在发送邮件时可以选择使用不同配置的数据库邮件。
- 多个账户。每个配置文件都可以包含多个故障转移账户。可以配置包含不同账户的不同配置文件，以跨多台电子邮件服务器分发电子邮件。
- 数据库邮件支持发送附件，附件大小是可以调控的。数据库邮件中使用 sysmail_configure_sp 存储过程，增强了对附件文件大小的可配置限制。
- 发送数据库邮件附件时，可以指定禁止的附件文件扩展名。使用 sysmail_configure_sp 系统存储过程在数据库邮件中设置一个禁止使用的文件扩展名列表。在列表中的文件扩展名将无法作为附件在数据库邮件中发送。

- 数据库邮件允许以 HTML 格式发送电子邮件。

除了这些特性外，数据库邮件在安全性上有很大的加强，默认情况下数据库是关闭的，视图特殊的数据库角色和权限才能使用数据库邮件，数据库邮件的活动可以记录到 SQL Server 日志或 Windows 应用程序事件日志中。发生数据库邮件时，系统会把数据库邮件的副本保存到 msdb 数据库中以便审核。

数据库邮件与 Service Broker 进行了结合，使用异步的方式传递。当调用 sp_send_dbmail 发送邮件时，数据库邮件向 Service Broker 队列中添加一个请求并立即返回，数据库邮件进程将邮件发送到指定地址。外部电子邮件组件接收请求并传递电子邮件。发送邮件的过程如图 17.19 所示。



17.19　数据库邮件体系结构

当用户执行 sp_send_dbmail 发送邮件时，存储过程向邮件对应的 Service Broker 队列中插入一项，并创建一条包含该电子邮件信息的记录。在邮件队列中插入新项将启动数据库邮件外部进程。该进程会读取数据库邮件配置，将电子邮件发送到相应的一台或多台电子邮件服务器。该外部进程还会在状态队列中插入一项，指示发送操作的结果。在状态队列中插入新项，将启动内部存储过程用于更新电子邮件信息的状态。

数据库邮件会把电子邮件信息保存到数据库中，包括邮件中的附件。数据库邮件视图提供了供排除故障使用的邮件状态，使用存储过程可以对数据库邮件队列进行管理。

17.3.2　配置数据库邮件

SQL Server 提供了数据库邮件配置向导，用于轻松实现数据库邮件的配置。在 SQL

Server 中配置数据库邮件的操作如下。

（1）在 SSMS 的对象资源管理器中展开“管理”节点，选择其下的“数据库邮件”子节点，在弹出的快捷菜单中选择“配置数据库邮件”选项，打开数据库邮件配置向导对话框。

（2）在向导的欢迎界面中单击“下一步”按钮，进入选择配置任务对话框，系统提供的配置任务如图 17.20 所示。

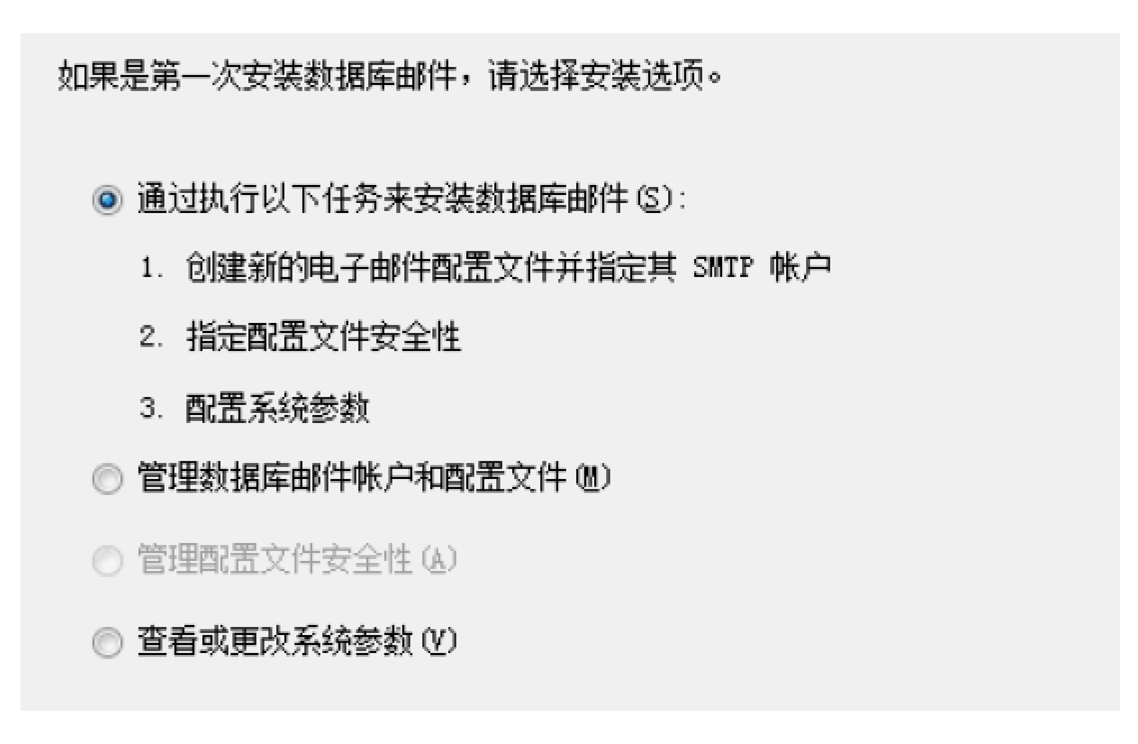


图 17.20　数据库邮件配置任务

（3）由于这里是第一次配置数据库邮件，所以选择第一个选项，单击“下一步”按钮。出于安全考虑，默认情况下数据库邮件是被禁用的，所以系统会弹出提示“数据库邮件功能不可用，是否启用此功能？”，在提示对话框中单击“是”按钮，启用数据库邮件并进入新建配置文件界面，如图 17.21 所示。

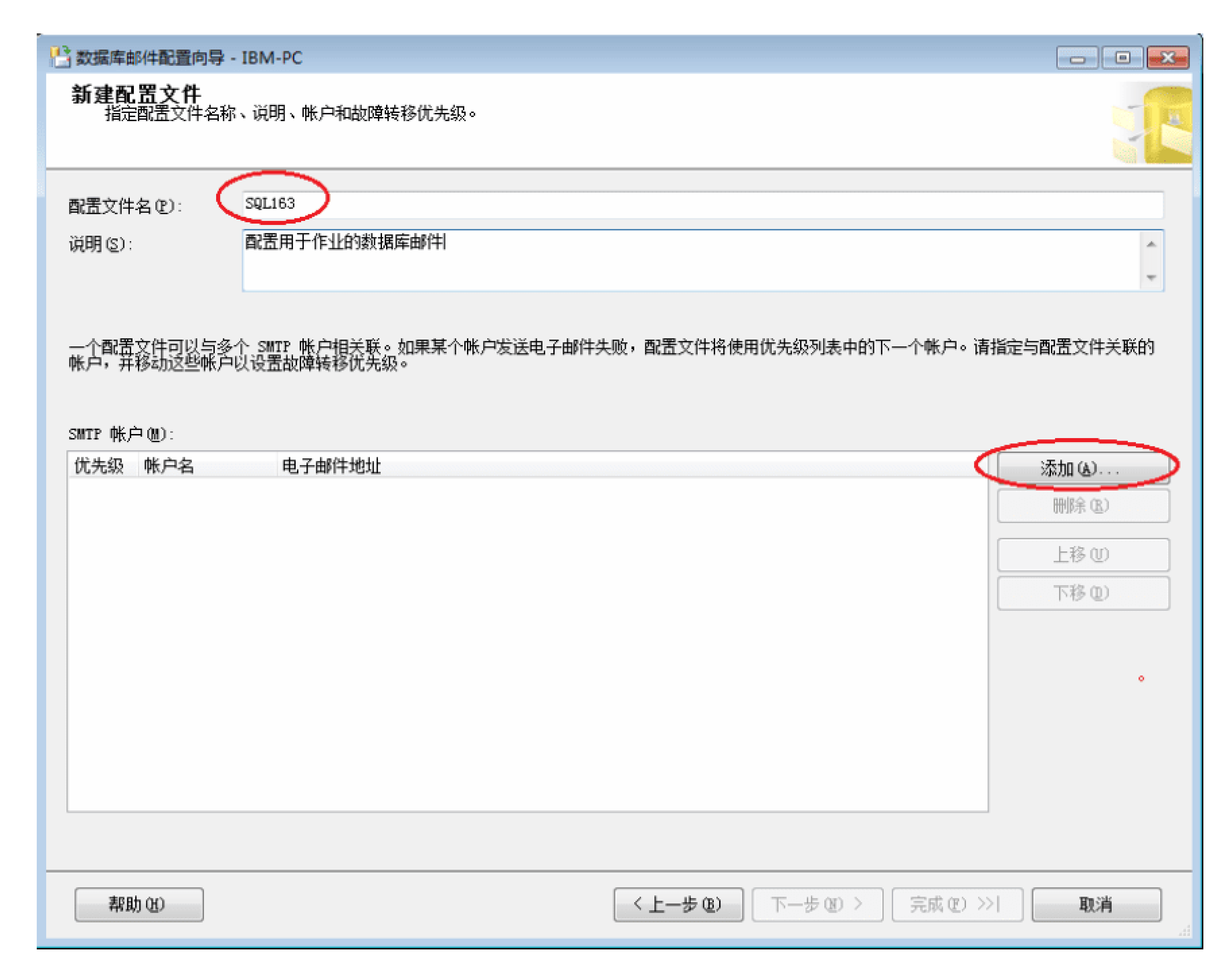


图 17.21　新建配置文件

（4）在“配置文件名”文本框中输入新建的配置文件名，“说明”文本框可以输入一些描述，也可以不输入。单击“添加”按钮，弹出“新建数据库邮件账户”对话框，如图 17.22 所示。

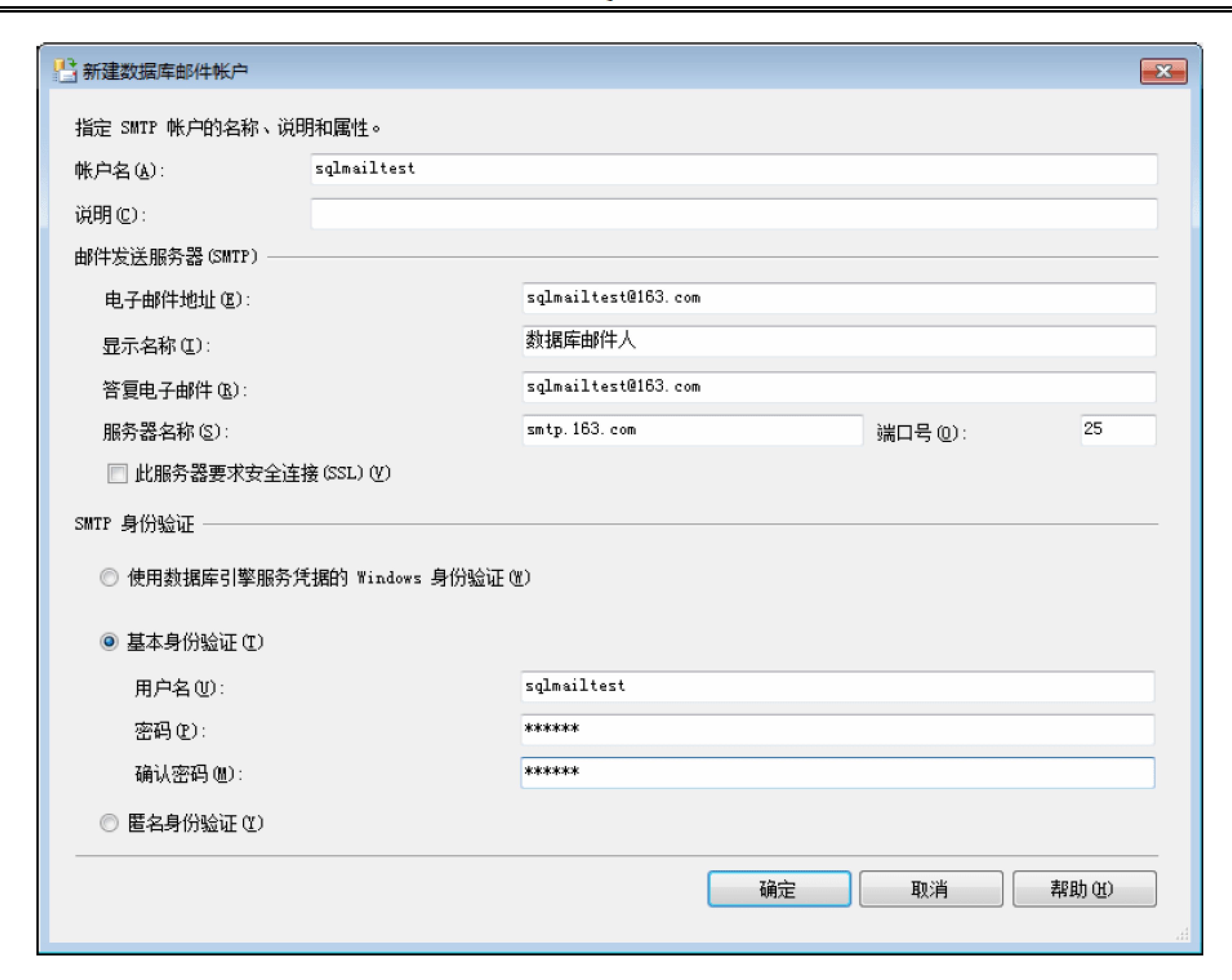


图 17.22　“新建数据库邮件账户”对话框

（5）这里笔者使用网易的邮件系统作为数据库邮件的邮件服务器，各种配置与一般的配置 Outlook 等邮件客户端并没有什么不同。配置完成后单击“确定”按钮回到向导对话框，在向导窗口中单击“下一步”按钮进入“管理配置文件安全性”对话框，如图 17.23 所示。

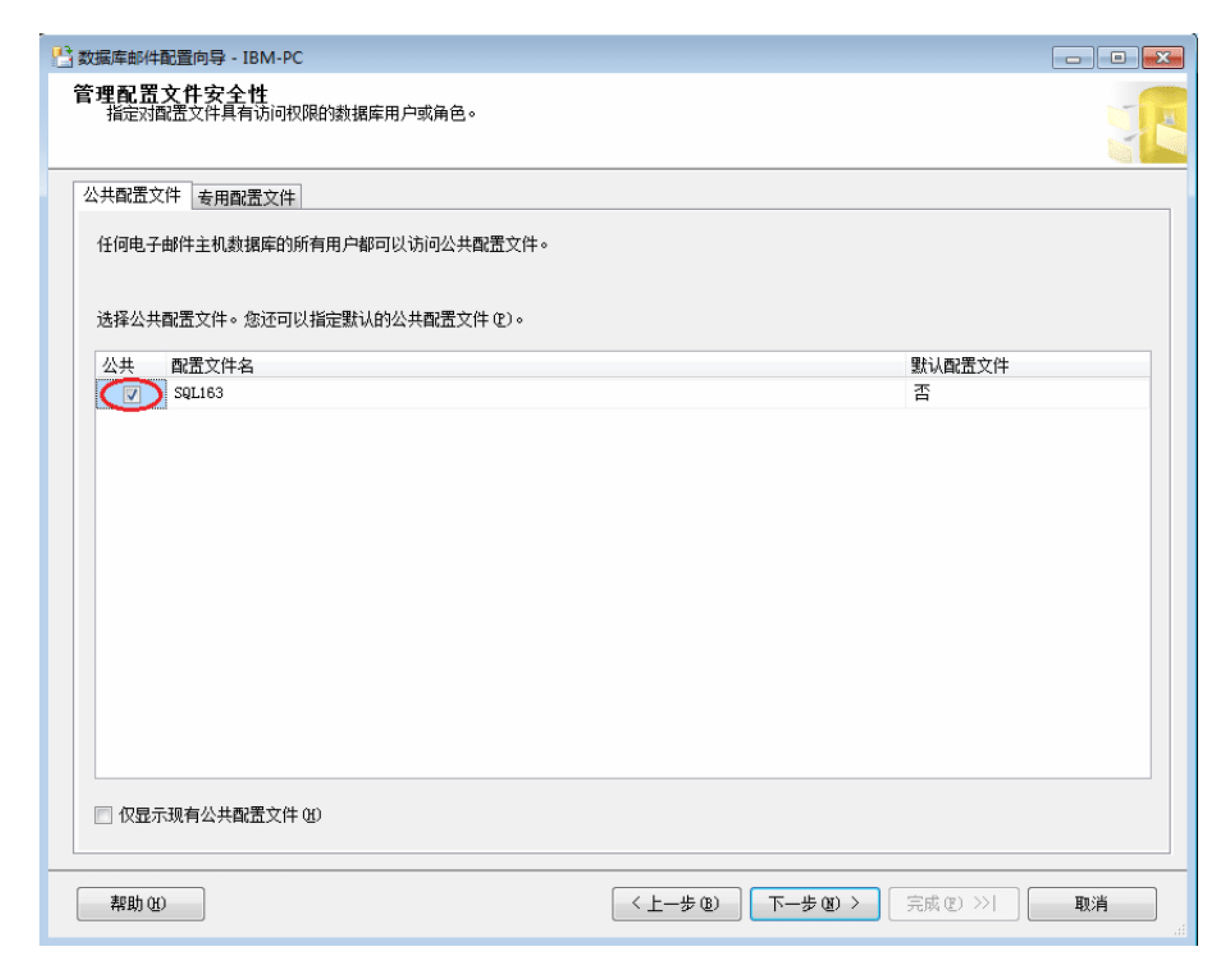


图 17.23　“管理配置文件安全性”对话框

（6）公共配置文件是指当前的配置可以被数据库中的所有用户访问，而专用配置文件就是指配置文件仅限于某个用户访问。这里配置为公共配置文件，选中“公共”复选框，然后单击“下一步”按钮，进入系统参数配置对话框，如图 17.24 所示。

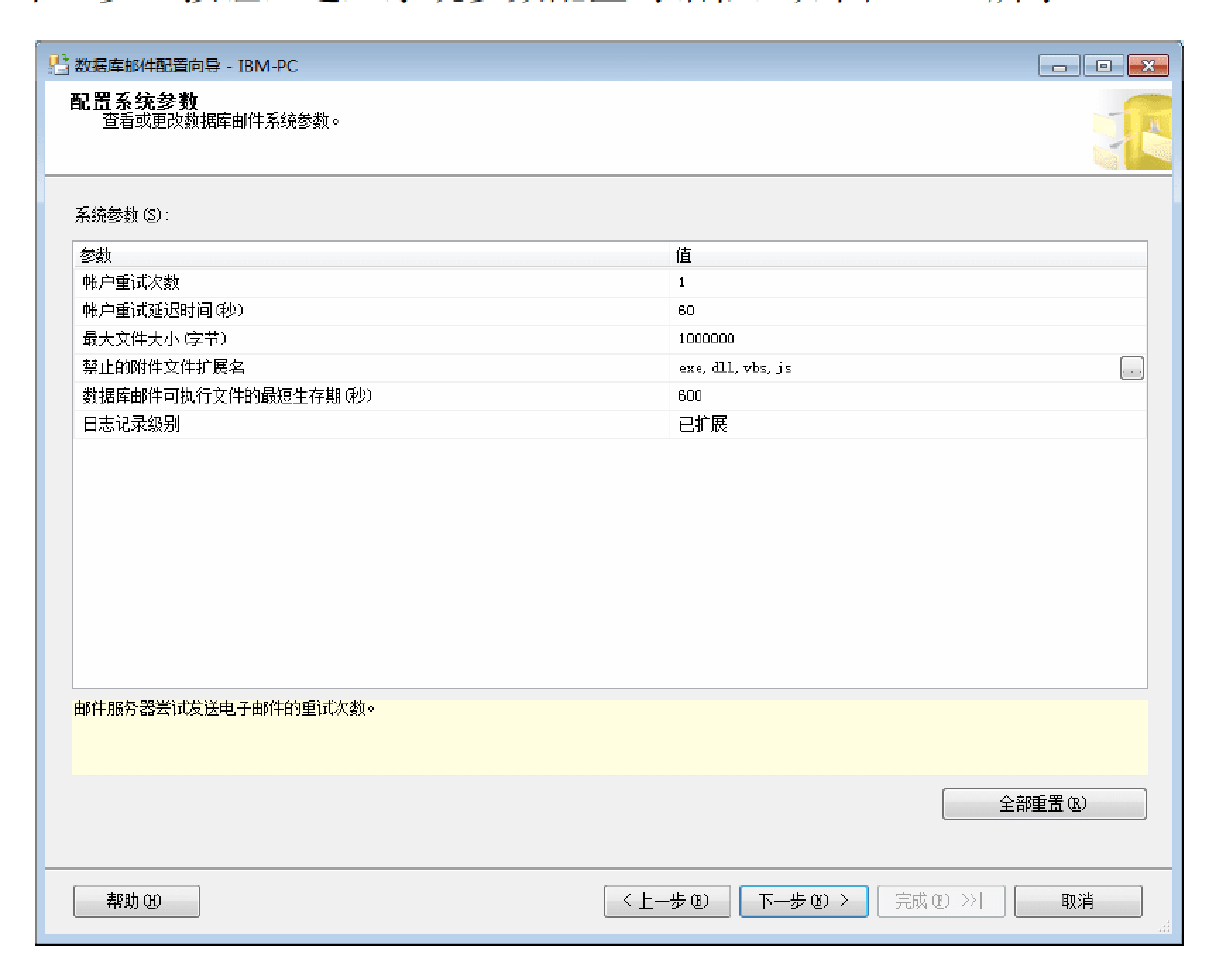


图 17.24　“配置系统参数”对话框

（7）“配置系统参数”对话框提供了重试次数、重试延迟时间、最大文件大小、禁止的附加扩展名、可执行文件最短生存期和日志记录级别几个参数选项。根据具体的需要修改系统参数，如果没有特殊要求则使用默认值即可。

（8）单击“下一步”按钮，最后再单击“完成”按钮即可完成向导，SQL Server 将根据向导中的配置完成对数据库邮件的配置。

（9）回到 SSMS，在对象资源管理器中选择“数据库邮件”节点，在弹出的快捷菜单中选择“发送测试电子邮件”选项，弹出发送测试电子邮件对话框，如图 17.25 所示。

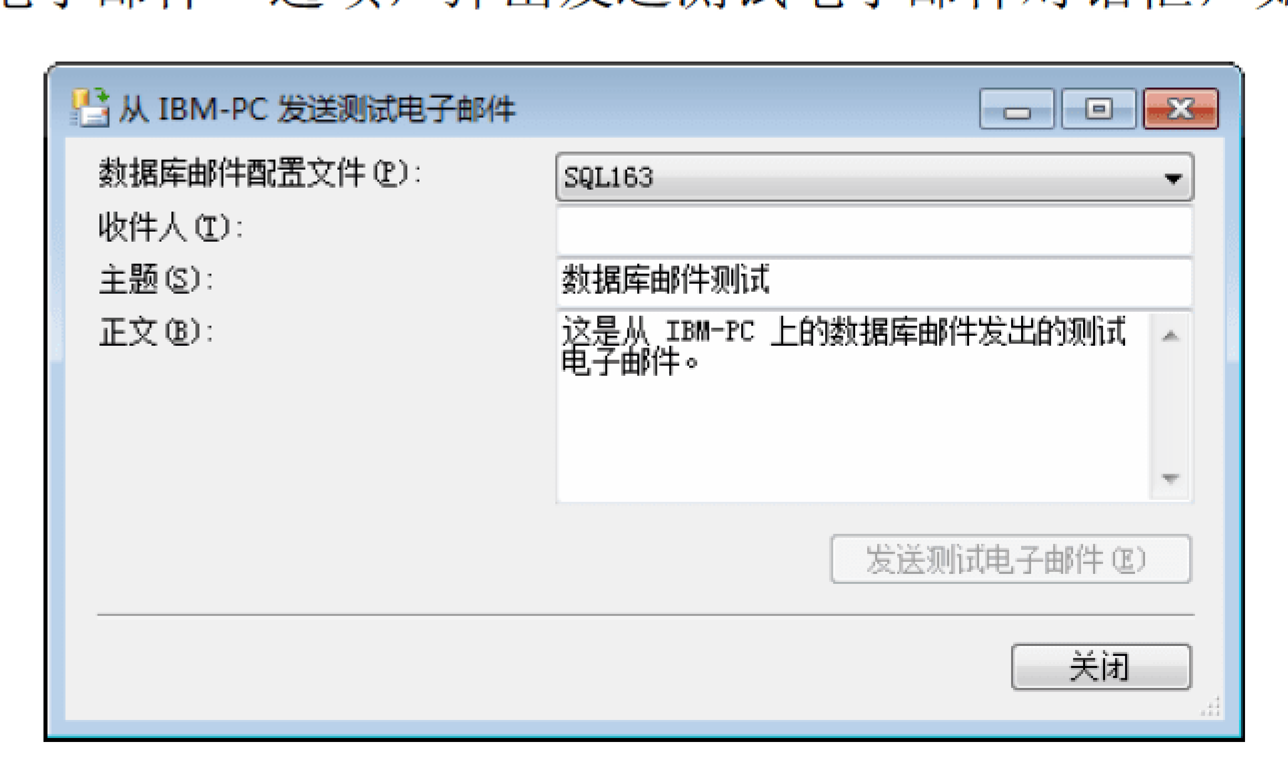


图 17.25　发送测试电子邮件

（10）在“收件人”文本框中输入收件人地址，然后单击“发送测试电子邮件”按钮，系统将使用 SQL163 配置中的邮件服务器发送测试邮件。如果收件人收到了电子邮件，则

说明数据库邮件配置成功；若未收到邮件，选择“数据库邮件”节点，在弹出的快捷菜单中选择“查看数据库邮件日志”选项，日志中将记录邮件发送失败的原因，根据具体错误提示，仍然使用数据库邮件配置向导修改数据库邮件的设置。

17.3.3　如何使用数据库邮件

配置好数据库邮件后便可调用系统存储过程 sp_send_dbmail，向指定收件人发送电子邮件了。该邮件可能包含查询结果集和文件附件。将邮件成功放入数据库邮件队列中后，sp_send_dbmail 将返回消息的 mailitem_id。例如要向一个用户发送一封邮件，则对应的脚本如代码 17.7 所示。

代码 17.7　使用数据库邮件来发送邮件

```
declare @mailid int
EXECUTE [msdb].[dbo].[sp_send_dbmail]
   @profile_name = 'SQL163'
   ,@recipients = 'sdfasdf@gmail.com'
   ,@body = '这是数据库邮件发出的测试电子邮件的正文。'
   ,@subject = '数据库邮件标题'
   ,@mailitem_id = @mailid OUTPUT
select @mailid
```

使用数据库邮件还可以将 SQL 查询的结果通过邮件发送给用户。例如，要查询 AdventureWorks2012 数据库的 Person.AddressType 表中的所有数据，并将该表所有数据发送给用户，则对应的 SQL 脚本如代码 17.8 所示。

代码 17.8　将查询结果发送邮件

```
declare @mailid int
EXECUTE [msdb].[dbo].[sp_send_dbmail]
   @profile_name = 'SQL163'
   ,@recipients = 'ssss@gmail.com'
   ,@query ='SELECT * FROM AdventureWorks.Person.AddressType' --SQL 查询
   ,@body = '这是数据库邮件发出的测试电子邮件的正文。'
   ,@subject = '数据库邮件标题'
   ,@mailitem_id = @mailid OUTPUT
select @mailid
```

这样用户将收到具有 Person.AddressType 表所有数据的邮件。

数据库邮件除了可以在存储过程中调用外，还可以配置在 SQL Server 代理中，在数据库作业完成或者出错等情况下向指定用户发送信息，具体配置和用法将在 17.4.3 节中讲解。

17.4　数据库警报

在数据库执行操作时发生错误的情况下，SQL Server 可以通过数据库警报将发送的错误通知数据库管理员，从而实现自动化的管理。另外，在锁定时间过长、日志文件过大等情况下发生的事件也可以通过警报的方式通知管理员。本节将主要讲解 SQL Server 代理中

的数据库警报。

17.4.1　创建操作员

数据库警报在发生时需要将警报发送给指定的用户，这些用户在 SQL Server 代理中就叫做操作员。操作员与数据库用户及 Windows 用户并没有任何关系，它其实更类似于通信簿中的联系人。可以为操作员指定姓名、电子邮件地址、Net send 地址、寻呼电子邮件地址和寻呼值班计划等信息。在 SSMS 中创建操作员的步骤如下。

（1）在 SSMS 的对象资源管理器中展开“SQL Server 代理”节点，选择其下的“操作员”节点，在弹出的快捷菜单中选择“新建操作员”选项，弹出“新建操作员”对话框，如图 17.26 所示。

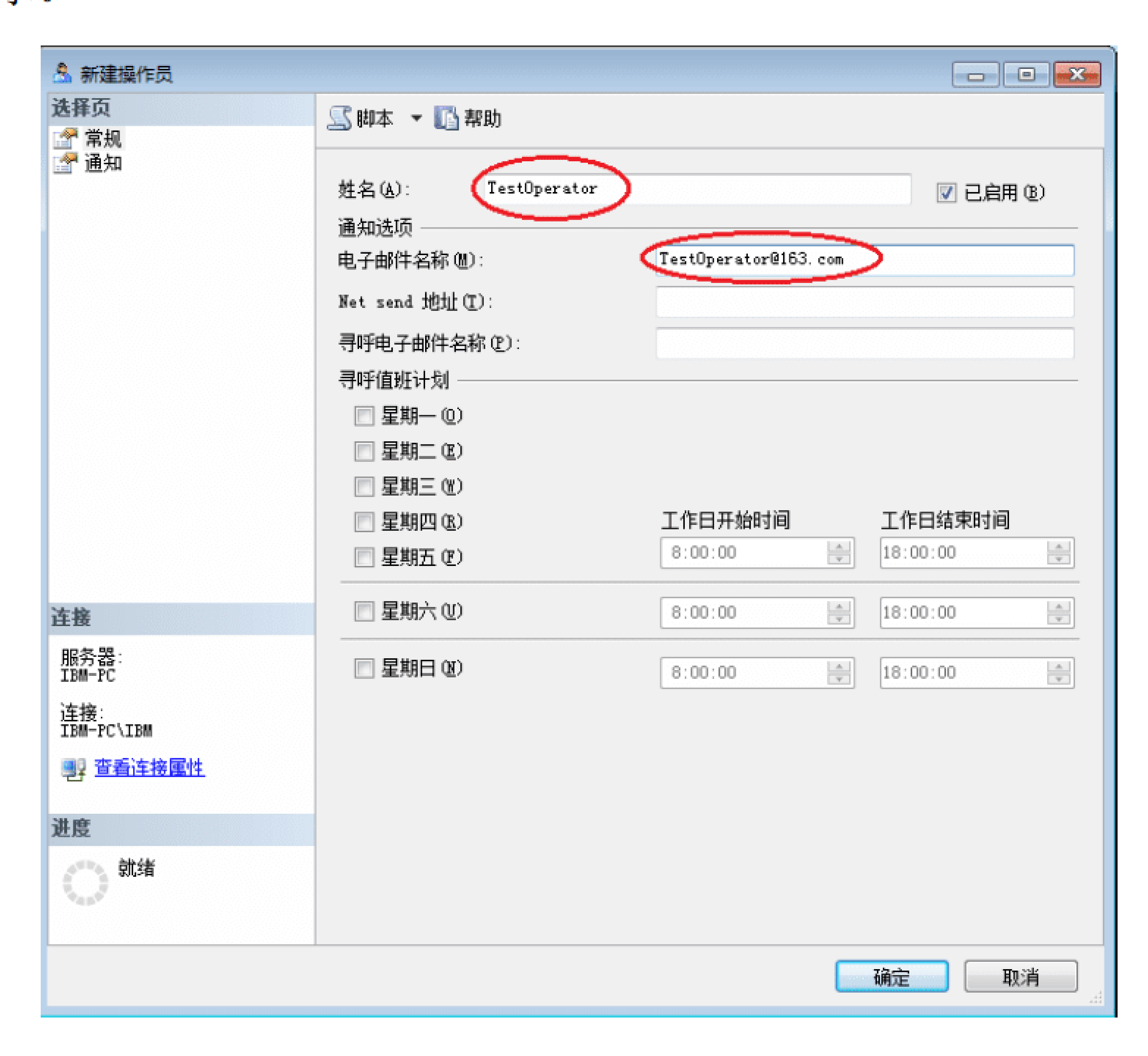


图 17.26 “新建操作员”对话框

（2）在“姓名”文本框中输入操作员的姓名，在 SQL Server 实例中，操作员名称必须是唯一的，并且长度不得超过 128 个字符。旁边的“已启用”复选框选中则表示该操作员在创建后便是可用的，取消选中则表示该操作员创建后是被禁用的。

（3）“通知选项”选项区域中提供了电子邮件名称、Net send 地址和寻呼电子邮件名称 3 种选项。如果使用电子邮件的方式通知操作员，则需要配置电子邮件名称。Net send 通知方式通过 Net send 命令向操作员发送消息，对于 Net send，需要指定网络消息的收件人。寻呼是通过电子邮件实现的。对于寻呼通知，需要提供操作员接收寻呼消息的电子邮件地址。若要设置寻呼通知，必须在邮件服务器上安装软件，处理入站邮件并将其转换为寻呼消息。寻呼通知的方式很少采用，这里不再详细介绍。

此处使用电子邮件的方式通知操作员，所以只需要在“电子邮件名称”文本框中输入电子邮件地址即可。

（4）单击“确定”按钮完成操作员的创建工作。

创建好操作员后便可在对象资源管理器的操作员节点下看到当前的操作员。

17.4.2　创建警报

在 SQL Server 代理中，事件由 SQL Server 生成并被输入到 Microsoft Windows 应用程序日志中。在 SQL Server 代理中可以定义警报，SQL Server 代理读取应用程序日志，并将写入的事件与定义的警报进行比较，当 SQL Server 代理找到匹配项时，它将发出自动响应事件的警报。除了监视 SQL Server 事件以外，SQL Server 代理还监视性能条件和 WMI 事件。若要定义警报，需要指定：

- 警报的名称。
- 触发警报的事件或性能条件。
- SQL Server 代理响应事件或性能条件所执行的操作。

例如在 TestDB1 数据库中，如果发送了 SQL 语法错误的事件，则执行 ChangeStatus 作业，并且将事件内容以电子邮件的方式，发送警报给前面创建的操作员 TestOperator。在 SSMS 中创建该数据库警报的操作如下。

（1）在 SSMS 的对象资源管理器中展开“SQL Server 代理”节点。选择其下的“警报”节点，在弹出的快捷菜单中选择“新建警报”选项，打开“新建警报”对话框，如图 17.27 所示。

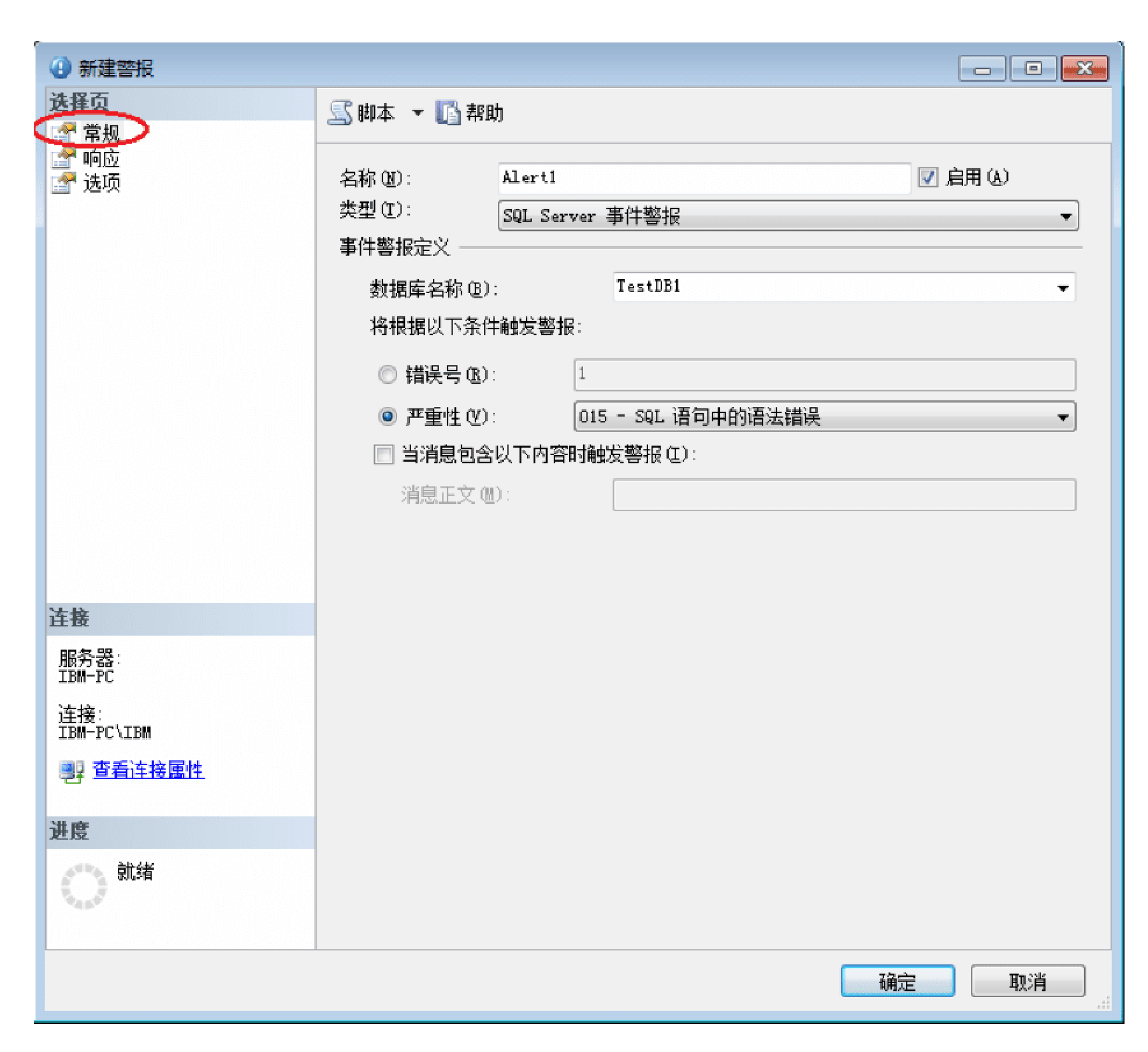


图 17.27　“新建警报”对话框

（2）在“名称”文本框中输入警报的名称，该名称在 SQL Server 实例内必须唯一，并且不能超过 128 个字符。旁边的“启用”复选框表示创建的警报是否启用。

（3）警报类型下拉列表框中提供了 SQL Server 事件警报、SQL Server 性能条件警报和 WMI 事件警报。如果选择警报类型为 SQL Server 事件警报，则需要指定：

- 数据库　SQL Server 代理仅在特定数据库中发生事件时才发出警报。
- 错误号　SQL Server 代理在发生特定错误时发出警报。
- 严重性　SQL Server 代理在发生特定级别的严重错误时发出警报。
- 消息正文　SQL Server 代理在指定事件的事件消息中包含特定文本字符串时发出警报。

其中数据库是必须定义的，错误号和严重性只能指定一个，而消息正文则是可选的参数。如果选择 SQL Server 性能条件警报，则对应的选项配置如图 17.28 所示。

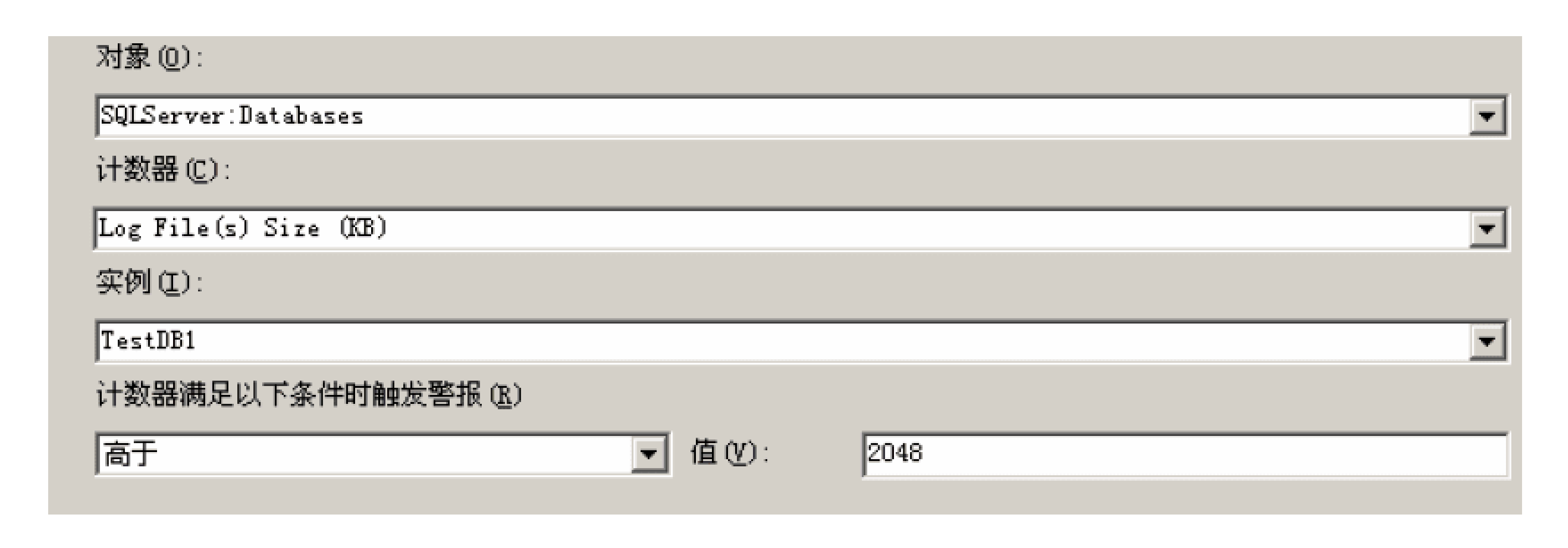


图 17.28　SQL Server 性能条件警报

图 17.28 中各个选项的含义如下：

- “对象”是要监视的性能区域。
- “计数器”是要监视的区域的属性。
- “实例”指定了要监视的属性的特定实例。
- “计数器满足以下条件时触发警报”和“值”则是对应计数器值的条件判断。

注意：性能计数器是周期采样的，SQL Server 代理使用它每隔 20 秒维护一次计数器，所以在达到阈值和发出警报之间的时间有所延迟。

如果选择 WMI 事件警报，则需要提供命名空间和 WMI 中的查询。这里要警报的是 SQL Server 事件，当 TestDB1 中发送严重性为 015 的事件时发送警报。

（4）选择“响应”选项页切换到警报响应界面，如图 17.29 所示。

（5）在发生警报时需要执行 ChangeStatus 作业，所以需要选中“执行作业”复选框，然后在作业下拉列表框中选择 ChangeStatus 选项。另外，还需要通过电子邮件通知 TestOperator 操作员，所以选中“通知操作员”复选框，然后在操作员列表中将 TestOperator 的电子邮件列选中。

（6）选择“选项”选项页，切换到警报选项配置界面，如图 17.30 所示。

（7）由于是向操作员发送电子邮件，所以在警报文本发送方式中选择“电子邮件”复选框。“要发送的其他通知消息”文本框中可以输入一些自定义的信息。在下面定义了“两次响应之间的延迟时间”，在延迟时间内再发生的警报将不会被响应。0 分钟 0 秒表示所有警报都将会被响应。

（8）单击“确定”按钮，完成新建警报的操作。

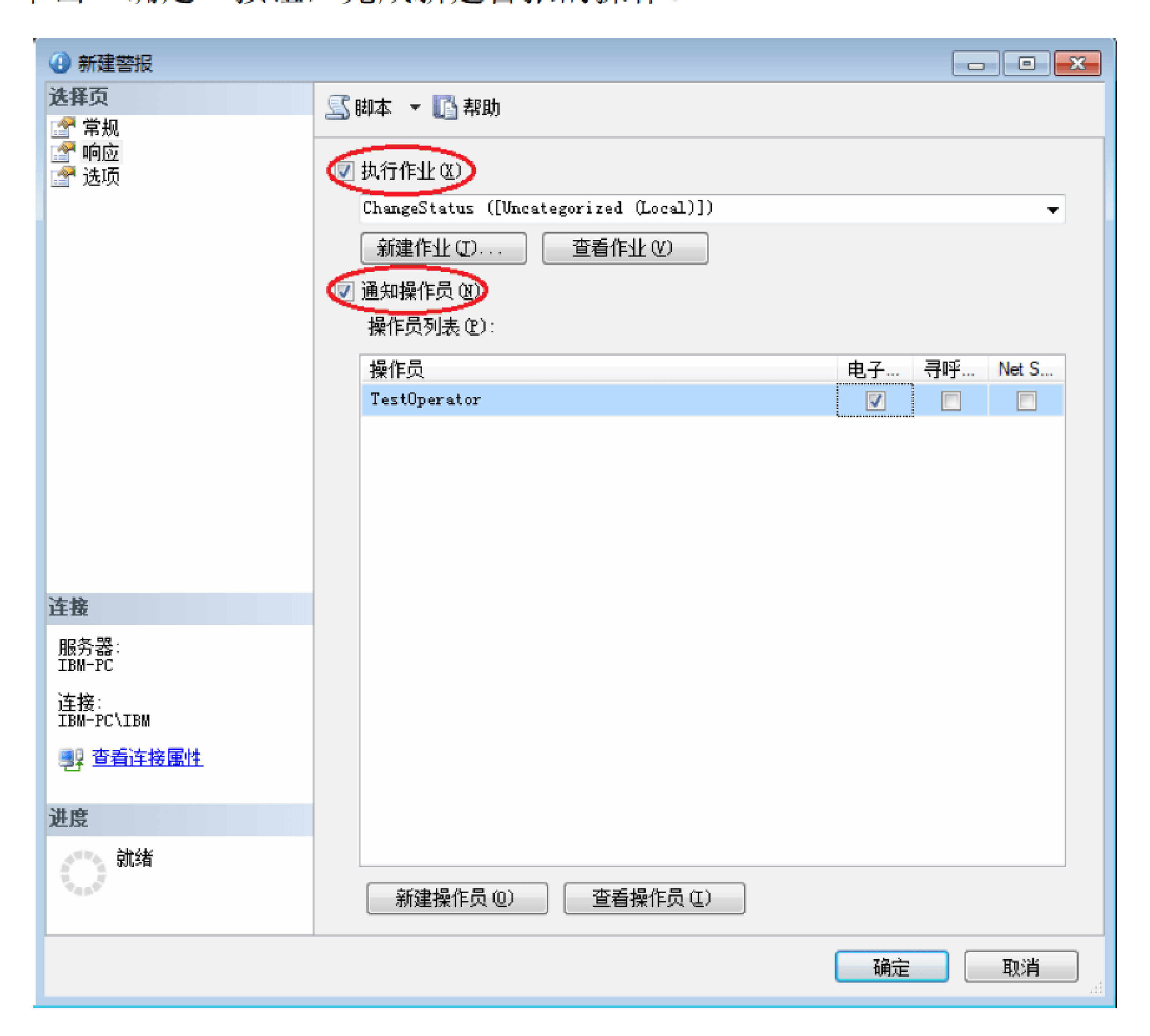


图 17.29　警报响应

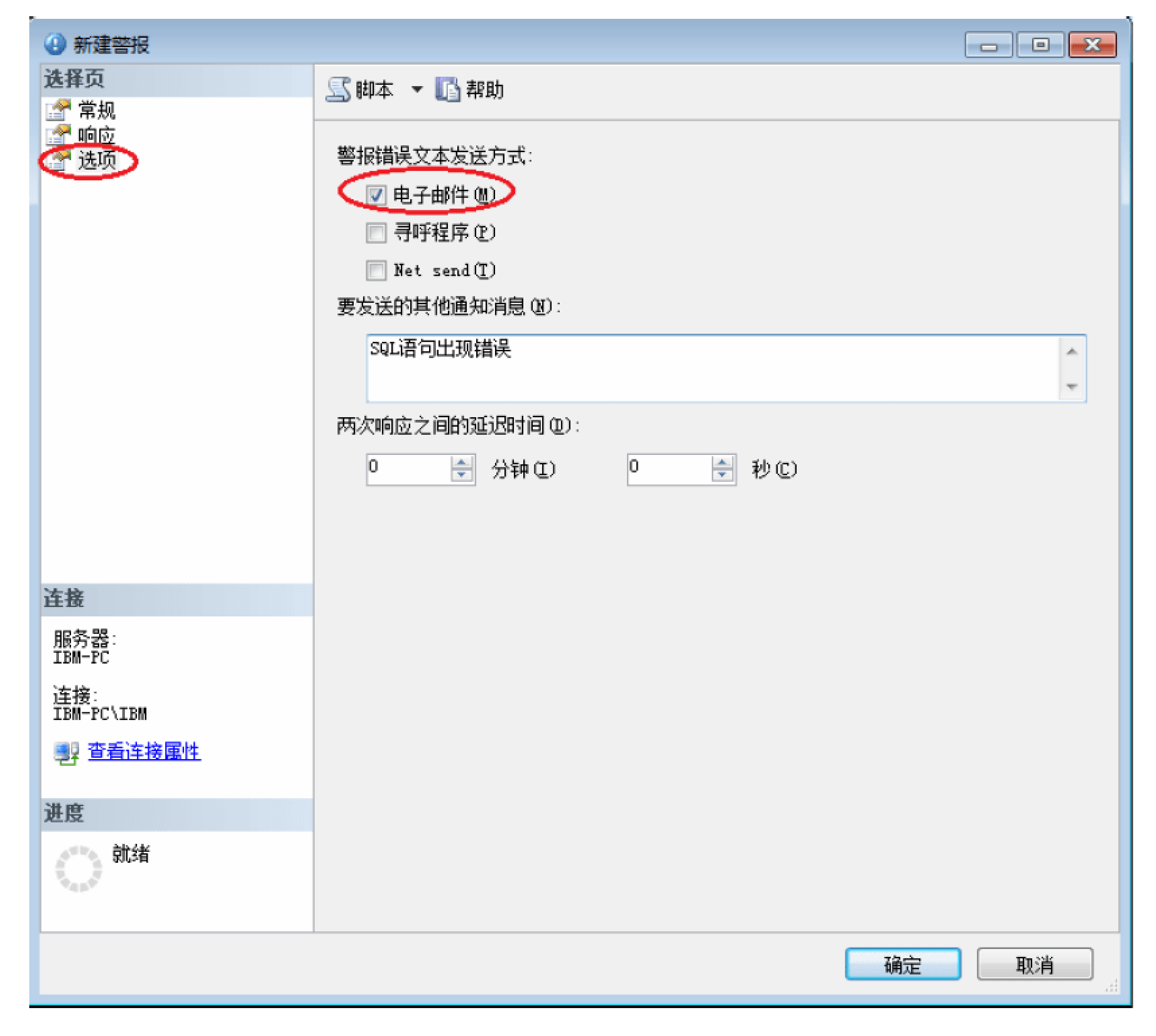


图 17.30　警报选项设置

17.4.3　为 SQL Server 代理配置数据库邮件

SQL Server 代理具有发送电子邮件的功能。可以通过配置 SQL Server 代理邮件，使其在出现下列情况时向预定义的操作员发送电子邮件：

- 警报触发时。可以配置警报，以针对所发生的特定事件发送电子邮件通知。例如，可以配置警报，将可能需要立即采取行动的特定数据库事件或操作系统情况通知操作员。有关配置警报的详细信息，请参阅定义警报。
- 计划任务成功完成或未完成（例如，数据库备份或复制事件）。例如，如果在月底的执行进程过程中出现错误，就可以使用 SQL Server 代理邮件通知操作员。

可以给一组收件人发送电子邮件消息，通知他们所计划作业的状态，以便用户采取可能的对策。例如，可以配置 SQL Server 代理，在备份作业完成时发送电子邮件。

默认情况下，SQL Server 代理邮件是关闭的。需要通过修改 SQL Server 代理属性来开启代理邮件功能，具体操作如下。

（1）在 SSMS 的对象资源管理器中选择“SQL Server 代理”节点，在弹出的快捷菜单中选择“属性”选项，打开 SQL Server 代理属性对话框。

（2）在属性对话框中选择“警报系统”选项页，切换到警报系统配置界面，如图 17.31 所示。

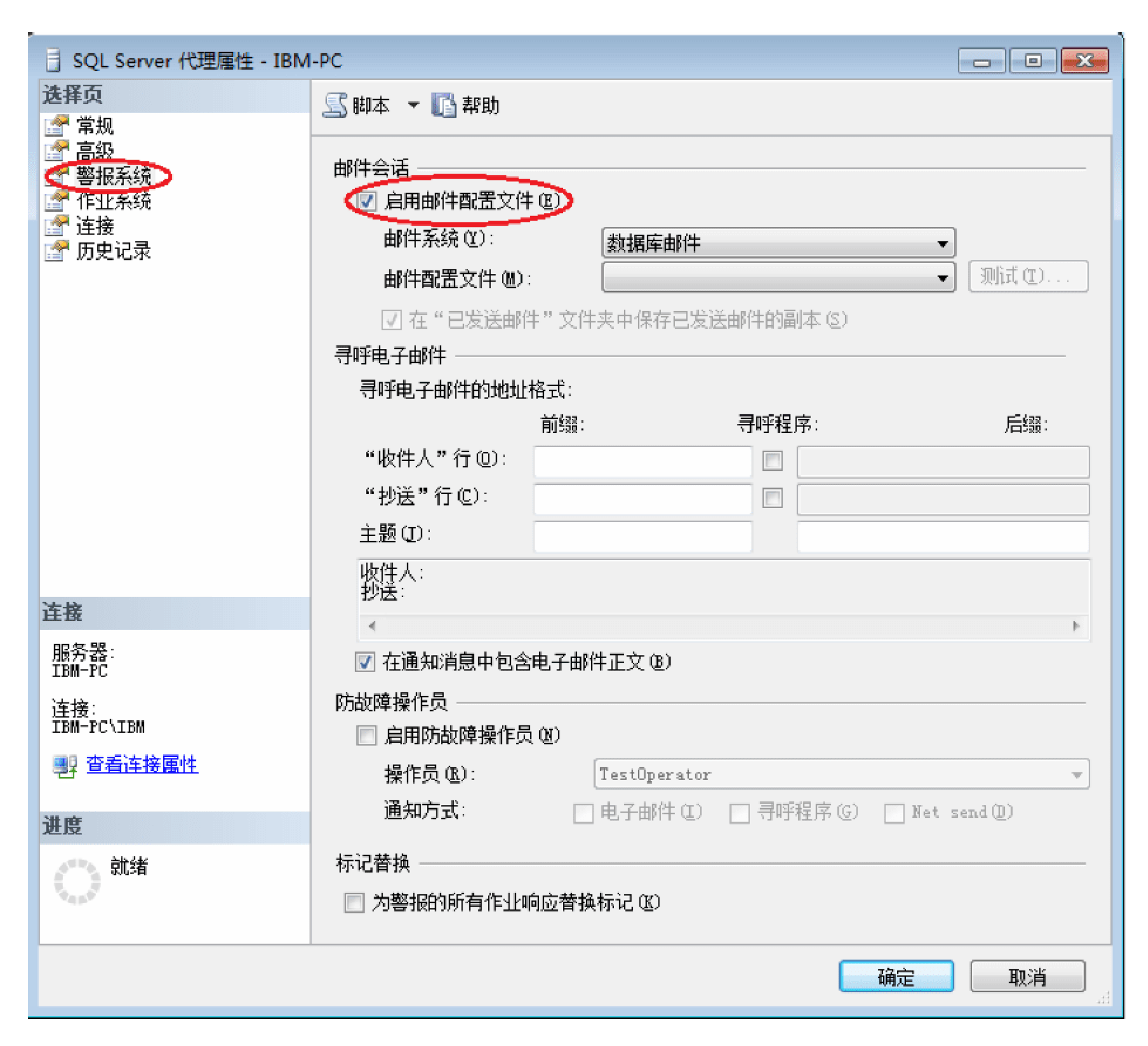


图 17.31　警报系统设置

（3）选中“启用邮件配置文件”复选框，表示在 SQL Server 代理中启用数据库邮件功能。在“邮件系统”下拉列表框只提供数据库邮件。选择“数据库邮件”选项，并在“邮件配置文件”下拉列表框中选中前面章节中创建的数据库邮件配置 SQL163。

注意：更改电子邮件设置后，必须重新启动 SQL Server 代理服务才能使更改生效。

至此，SQL Server 代理将利用配置的数据库邮件账户发送电子邮件，当警报发生时系统将发送电子邮件通知操作员。

17.4.4　为作业设置通知

在数据库作业成功完成或者执行失败的情况下，作业可以通过其中的通知功能向操作员报告它的执行情况。

作业的通知功能只是由该作业触发，而不像警报那样可以由其他模块触发。另外，作业中的通知功能只是一个报告操作员的功能，并不能在作业成功或者失败时像警报那样执行其他的数据库作业。在 SSMS 中，配置数据库作业的通知功能操作如下。

（1）在 SSMS 的对象资源管理器中右击需要设置通知的作业，在弹出的快捷菜单中选择“属性”选项，打开该作业的属性对话框。

（2）在属性对话框中选择“通知”选项页切换到通知配置界面，如图 17.32 所示。

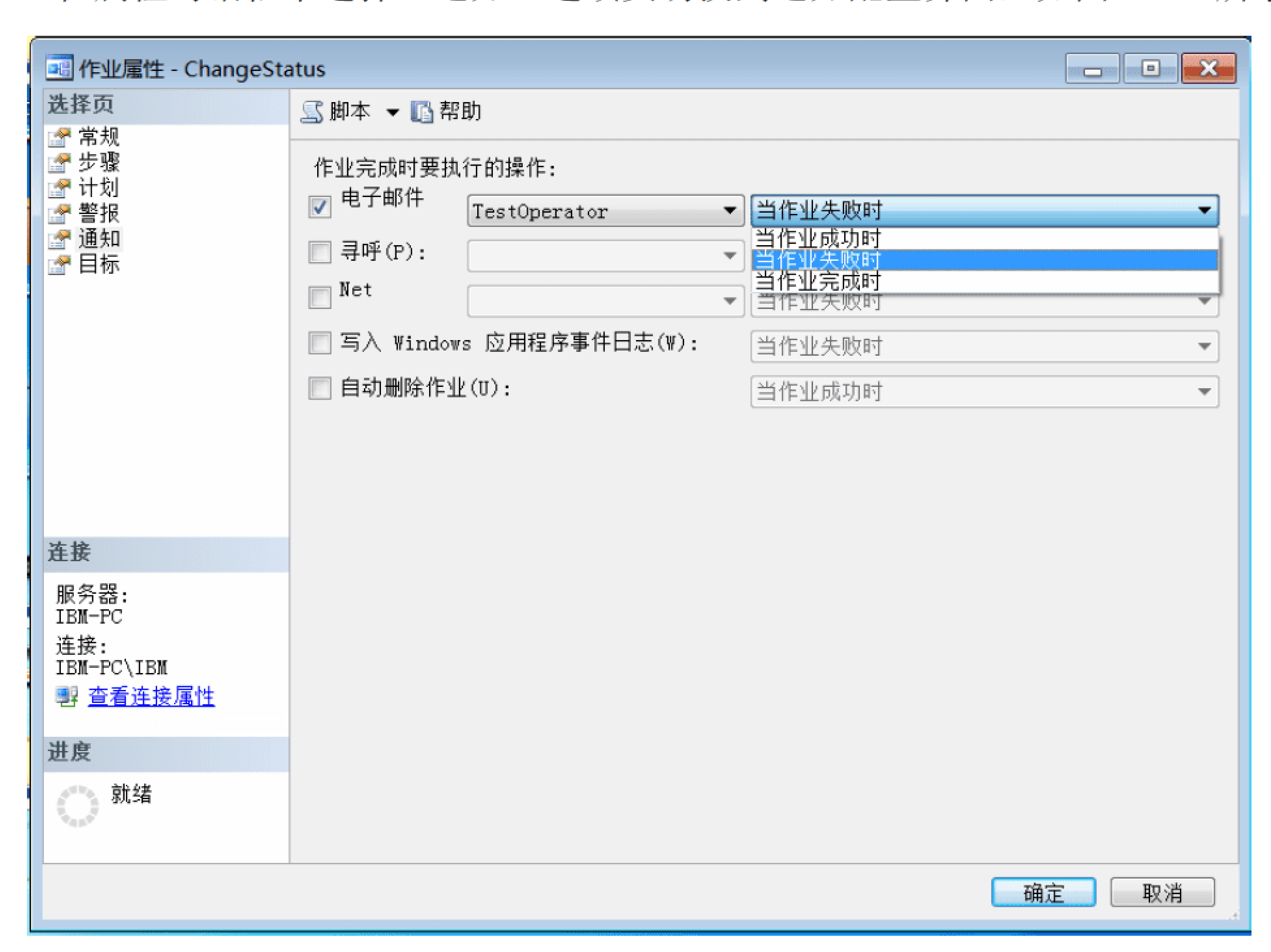


图 17.32　作业通知

（3）选中“电子邮件”复选框，在操作员下拉列表框中列出了所有的操作员，选择需要通知的操作员。接下来的事件下拉列表框中提供了当作业成功时、当作业失败时和当作业完成时 3 个选项，选择“当作业完成时”选项就是指无论作业是成功还是失败都会触发。

对于比较频繁而长期的数据库作业，一般并不需要在作业成功时通知操作员，只需要在作业失败时通知操作员即可。

（4）如果需要将作业执行的日志写入 Windows 应用程序事件日志中，则选中对应的复选框即可。

（5）如果希望作业在成功或者失败后将自身删除，则选中“自动删除作业”复选框。

（6）单击“确定”按钮，作业通知设置完成。

（7）如果选择的是在作业失败时通知操作员，则修改作业步骤，使得作业执行必然失败，然后运行该作业，不久后操作员将收到通知邮件，说明作业通知配置成功。

17.5 维护计划

维护计划可创建所需的任务工作流，以确保优化数据库、定期进行备份并确保数据库一致。维护计划与 SQL Server 代理紧密结合，维护计划会创建由 SQL Server 代理作业运行的 Integration Services 包。可以按预订的时间间隔手动或自动运行这些维护任务，从而大量简化数据库维护的配置工作。本节将主要介绍维护计划的使用。

17.5.1 维护计划向导

维护计划可以提供以下功能：

- 使用各种典型维护任务创建工作流的功能。此外，还可以创建自己的 Transact-SQL 脚本。
- 概念性层次结构。使用每个计划可创建或编辑任务工作流。每个计划的任务可分组到子计划中，可以安排这些子计划在不同时间运行。
- 支持可以用于主服务器和目标服务器环境中的多服务器计划。
- 支持在远程服务器上记录计划历史记录。
- 支持 Windows 身份验证和 SQL Server 身份验证，但建议尽可能使用 Windows 身份验证。

维护计划可以通过维护计划向导来完成。例如要创建一个数据库备份的维护计划，每天晚上 2：00 点对 TestDB1 数据库进行完整备份，则对应的操作如下。

（1）在 SSMS 的对象资源管理器中展开“管理”节点，选择其下的“维护计划”子节点。在弹出的快捷菜单中选择“维护计划向导”选项，弹出“维护计划向导”对话框。

（2）单击“下一步”按钮，进入选择计划属性界面，如图 17.33 所示。在“名称”文本框中输入计划的名称，“说明”文本框中可以输入对计划的说明。一个计划中可以包含多个项目任务，可以为每个项目任务单独设置计划，也可以统一使用一个计划。

（3）单击“更改”按钮，进入“新建作业计划”对话框，如图 17.34 所示。

该对话框与创建计划时对话框相同，将计划设置为每天晚上 2：00 点执行，单击“确定”按钮回到“维护计划向导”对话框。

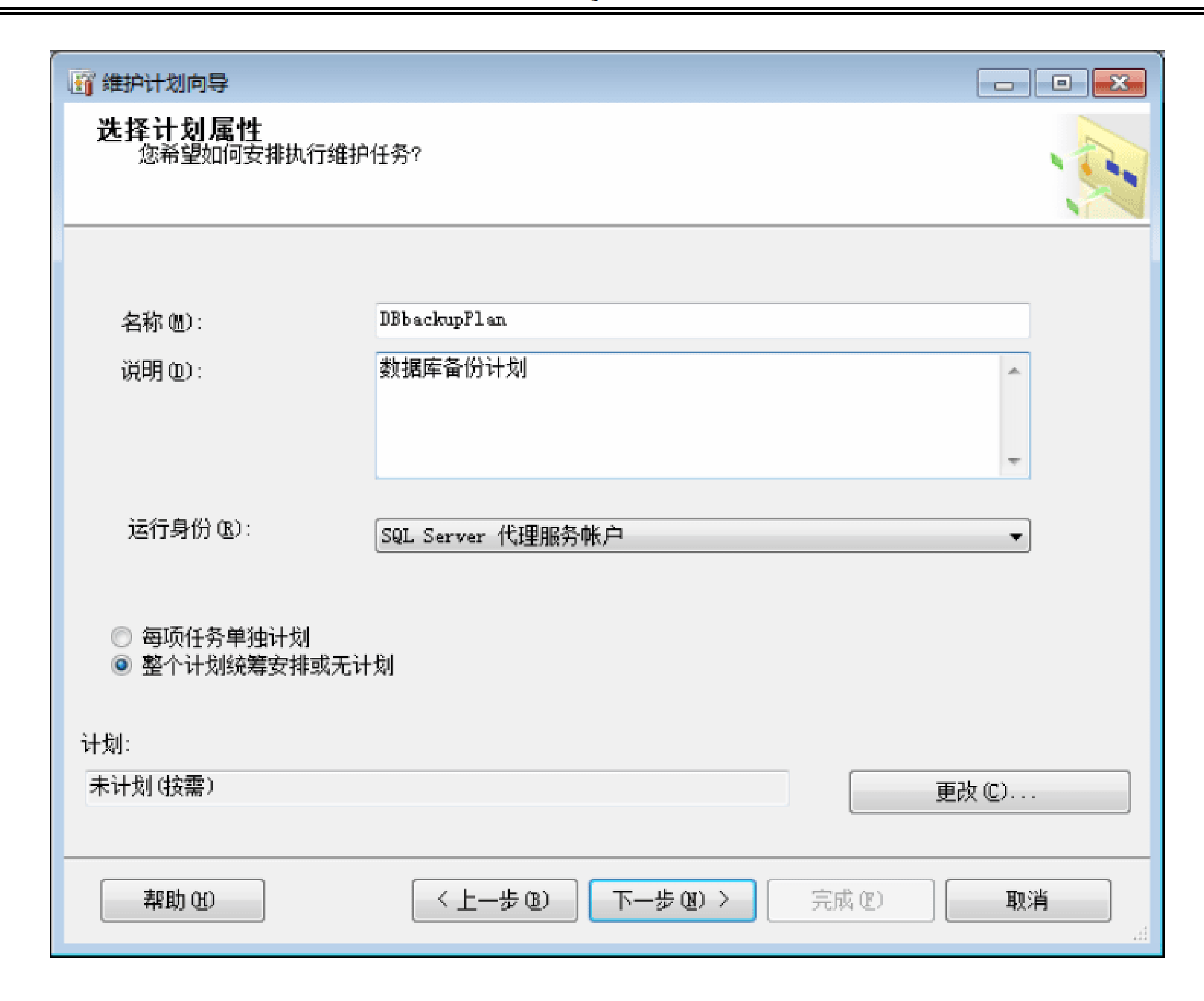


图 17.33　选择计划属性界面

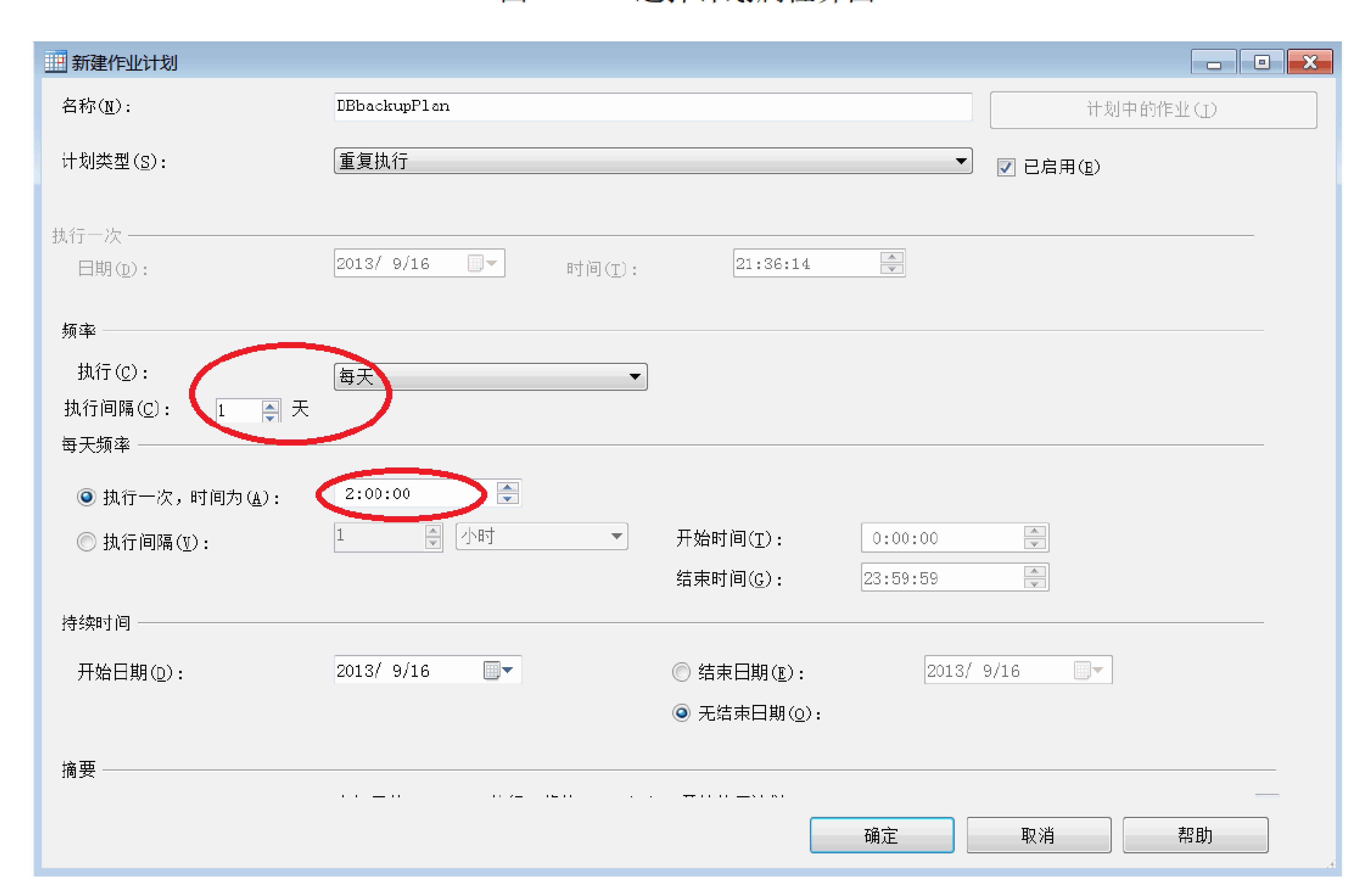


图 17.34　“新建作业计划”对话框

（4）单击“下一步”按钮，进入选择维护任务界面，如图 17.35 所示。这里列出了能够通过维护计划向导来完成的维护任务，这里执行数据库的完整备份，所以选择“备份数据库（完整）”复选框。

（5）单击“下一步”按钮，进入选择执行任务的顺序界面，这里是调整多个任务的执行顺序，由于只有一个任务，所以此处不进行任何修改。

（6）单击“下一步”按钮，进入备份数据库任务的设置界面，如图 17.36 所示。

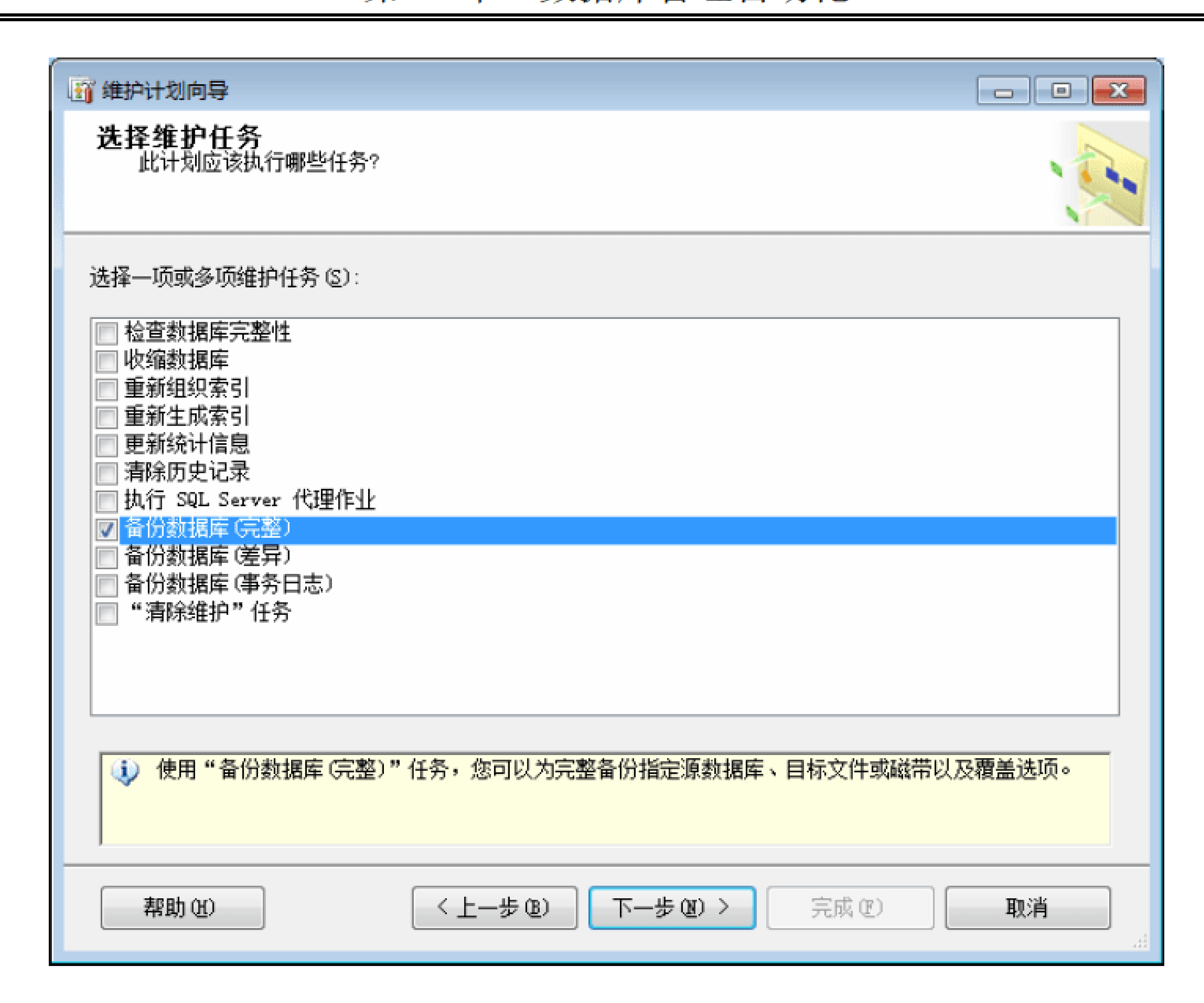


图 17.35　选择维护任务界面

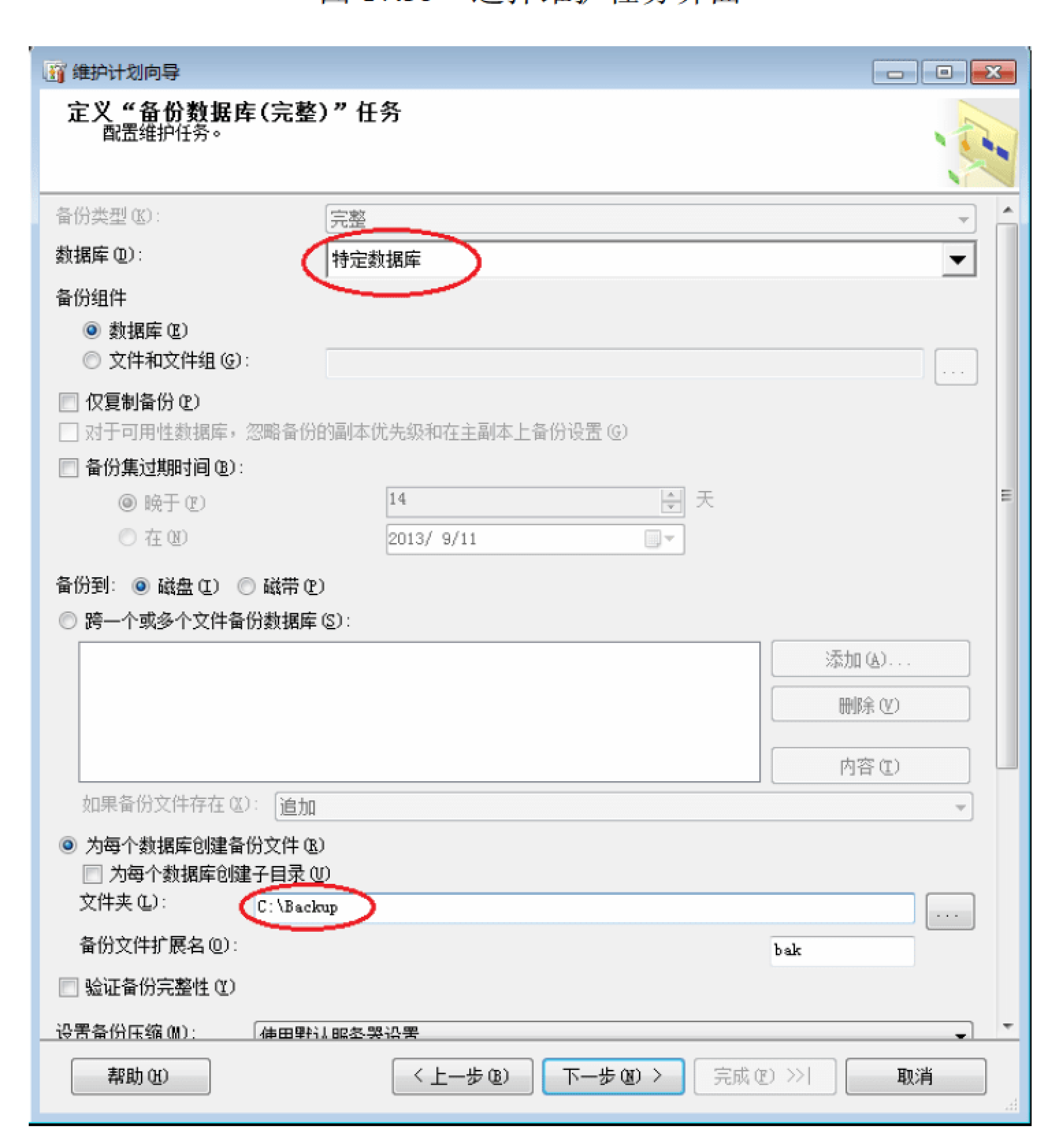


图 17.36　备份数据库任务设置

在数据库下拉列表框中选择要备份的数据库"TestDB1"（选中一个数据库后就会变成"特定数据库"），备份到磁盘上，这里采用为每个数据库创建备份文件的方式进行备份，备份文件夹设置为"C:\Backup"。

（7）单击"下一步"按钮，进入报告选项设置界面，如图 17.37 所示。备份数据库后

的报告可以以日志文件的形式保存在本地文件，也可以以电子邮件的形式发送给操作员。

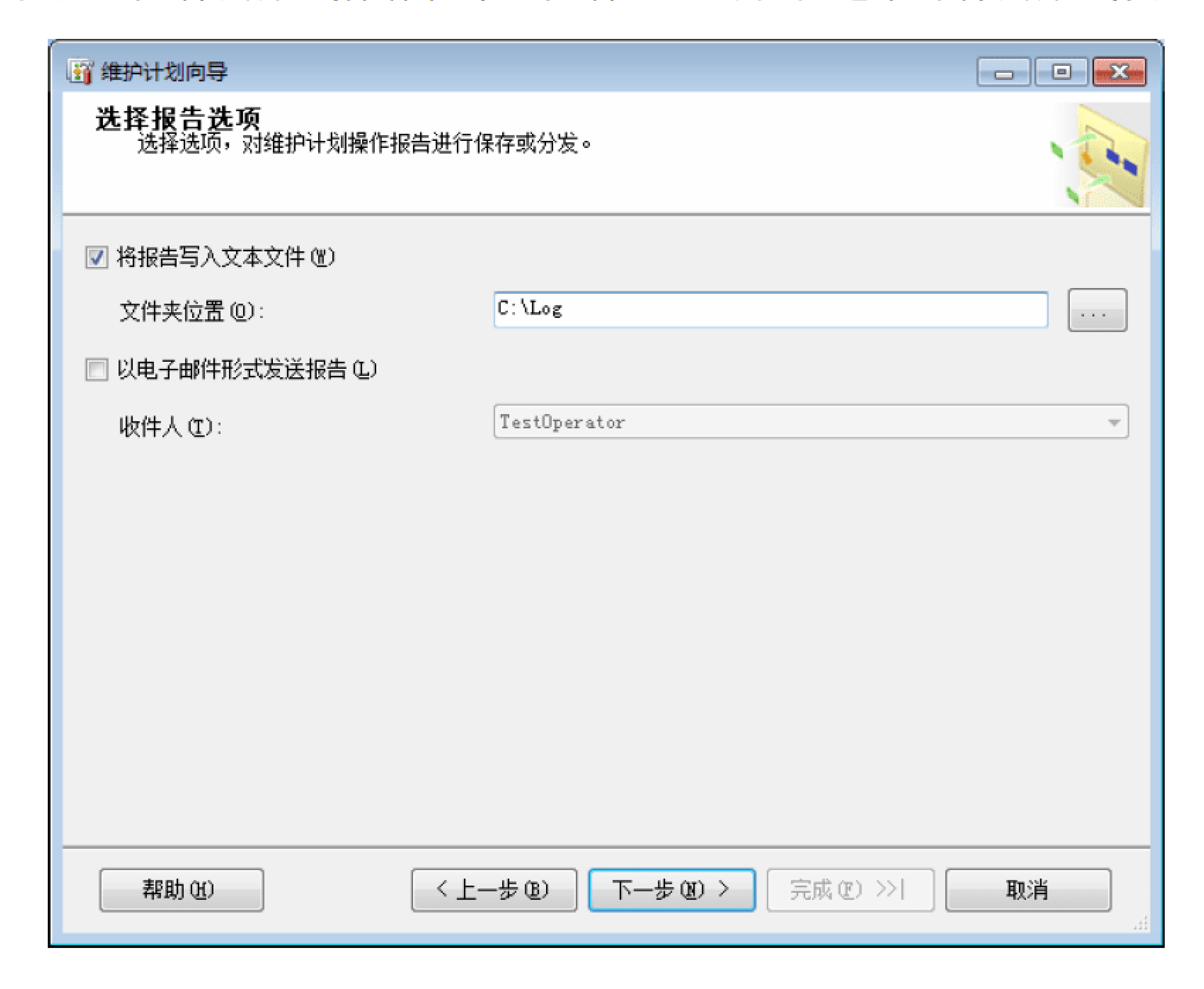


图 17.37　报告选项设置

（8）单击“下一步”按钮弹出向导完成的界面，最后再单击“完成”按钮，系统将开始根据向导中的配置创建维护计划。创建成功后在对象资源管理器中可以看到“维护计划”节点和“作业”节点下都多了个数据库备份的计划，如图 17.38 所示。

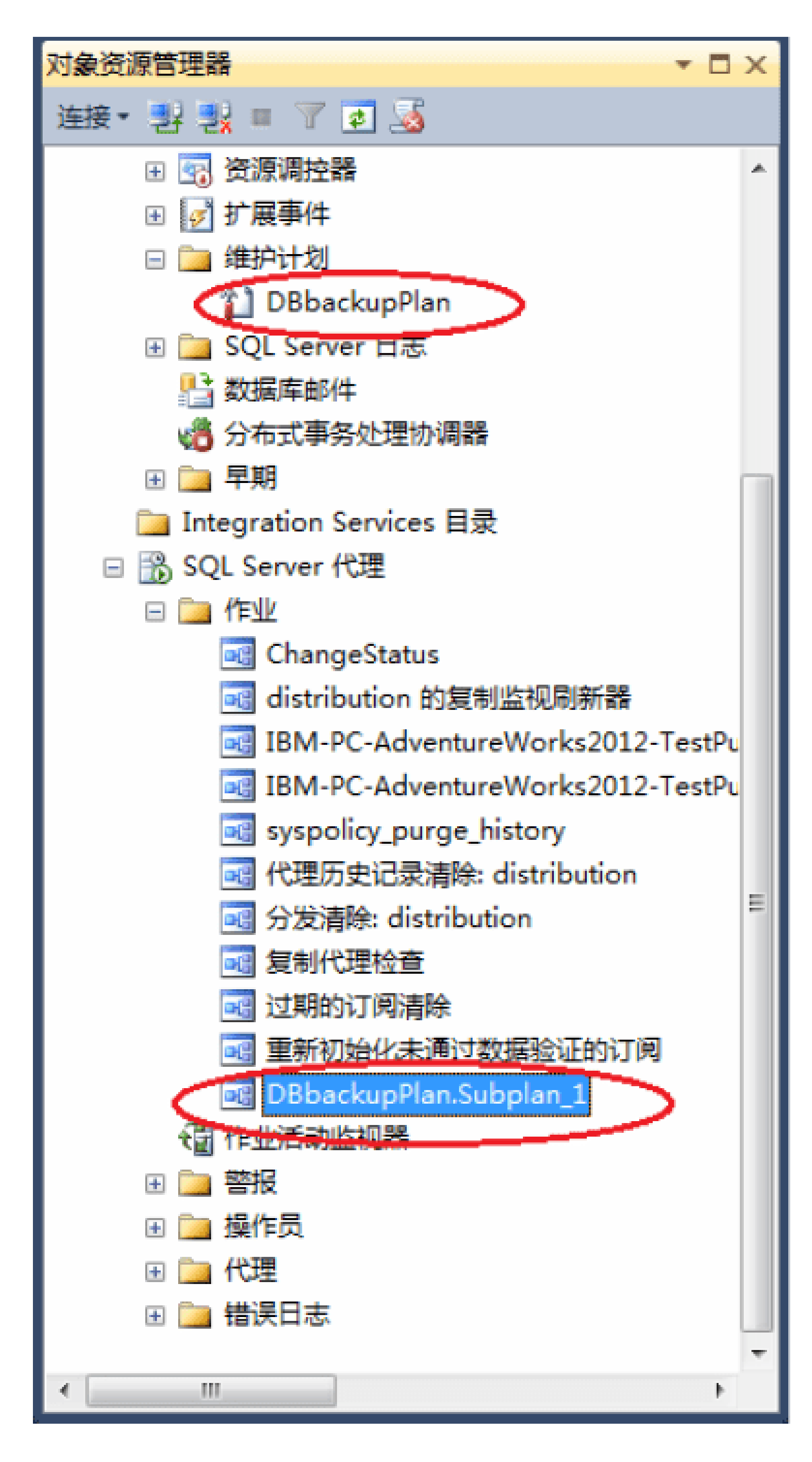


图 17.38　添加后的维护计划

17.5.2　配置维护计划

除了通过维护计划向导配置维护计划外，还可以通过在 SSMS 中拖曳控件的方式配置维护计划。

例如现在需要对数据库 TestDB1 进行收缩，再将数据库备份，然后通知操作员这一连串的操作，在维护计划中配置操作如下。

（1）在 SSMS 的对象资源管理器中展开“管理”节点，选择其下的“维护计划”子节点，在弹出的快捷菜单中选择“新建维护计划”选项，系统将提示输入维护计划名。

（2）在“计划名”文本框中输入 DBbackupPlan2，进入维护计划的设计界面，如图 17.39 所示。其中左侧是任务模块工具箱，右侧是属性配置，中下方则是维护计划设计的主面板。

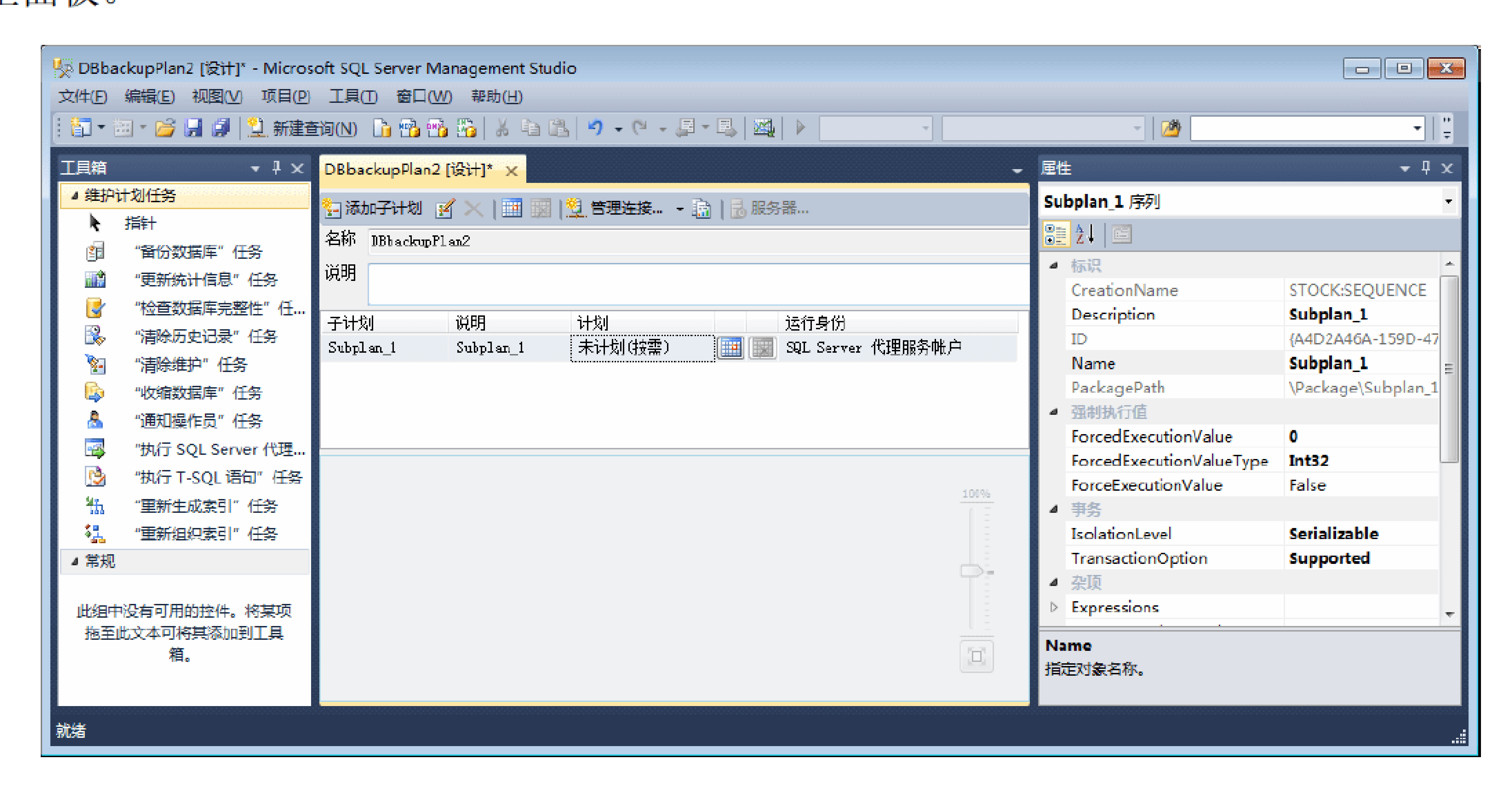


图 17.39　维护计划设计界面

（3）从工具箱中将需要的任务：收缩数据库任务、备份数据库任务和通知操作员任务拖曳到设计主面板中，如图 17.40 所示。

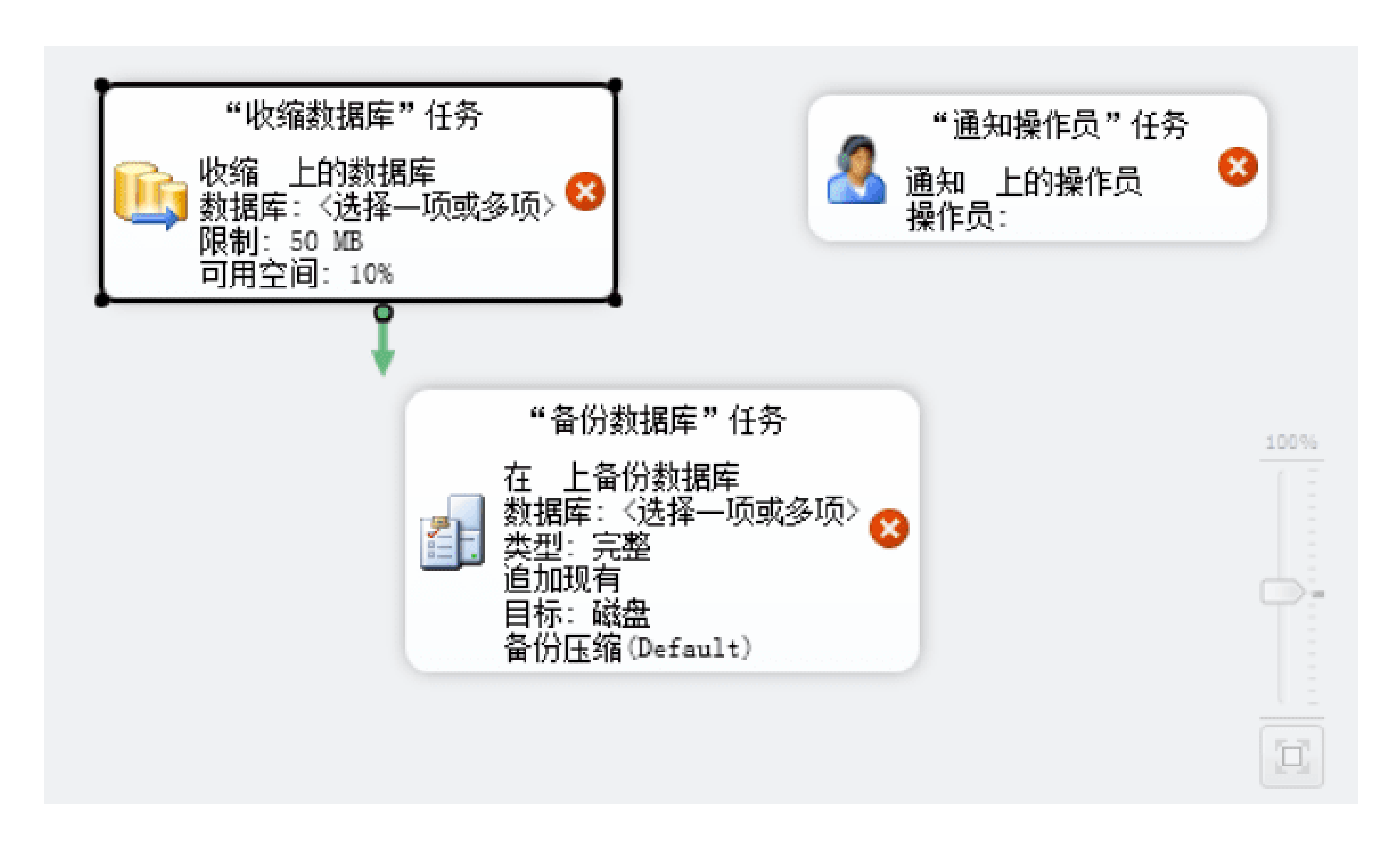


图 17.40　设计维护计划

（4）右击每个任务模块，在弹出的快捷菜单中选择“编辑”选项，配置每个任务的数据库、路径等属性，不同的任务模块有不同的配置界面。

注意：每个模块上的红色小叉表示该模块的属性有误，配置成为正确的属性后红色小叉才会消失。

（5）拖曳收缩数据库任务下的绿色箭头到备份数据库任务上，表示在完成收缩数据库操作后执行备份数据库操作。拖曳备份数据库任务下的绿色箭头到通知操作员任务上，表示在备份数据库任务完成后执行通知操作员任务。完成后的维护计划如图 17.41 所示。



图 17.41　设置任务工作流

（6）默认情况下任务之间的绿色箭头表示任务执行成功时进入下一个任务，如果要在任务失败的时候通知管理员，则右击备份数据库任务与通知管理员任务之间的绿色箭头，选择“编辑”选项，系统将弹出优先约束编辑器，如图 17.42 所示。

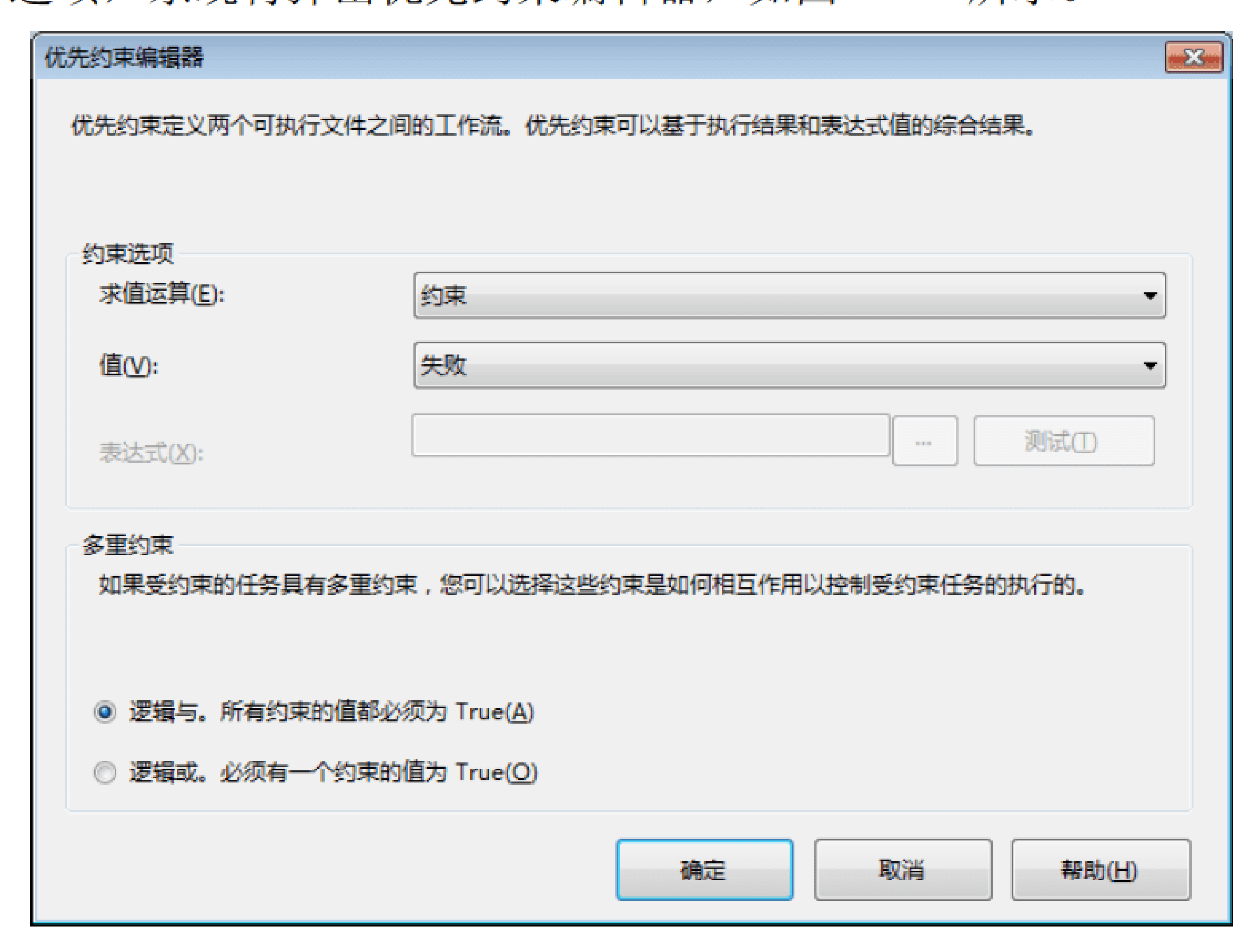


图 17.42　优先约束编辑器

（7）在“求值运算”下拉列表框中选择“约束”选项，“值”下拉列表框中选择“失败”选项，表示在失败的时候执行下一个任务。单击“确定”按钮回到维护计划设置界面，

原来的绿色箭头变成了红色箭头。

🔔**技巧**：可以不通过优先约束编辑器设置“失败”操作，只需要选中绿色箭头，然后在右侧的属性列表中选择 Value 属性为 Failure 即可。

（8）单击维护计划设计主面板上的子计划日历图标，弹出作业计划属性对话框。设置该作业计划为每天晚上 3：00 点执行一次，然后单击“确定”按钮回到设计主界面，如图 17.43 所示。



图 17.43 完成的维护计划

（9）单击工具栏的“保存”按钮，即可将该维护计划保存到数据库中，同样，在对象资源管理器中“维护计划”节点和“作业”节点下都可以看到该维护计划。

🔔**说明**：一个维护计划中可以包含多个子计划，并且可以为每个子计划指定数据库计划，每个子计划对应一个数据库作业。

17.5.3 维护计划管理

维护计划最终以 SSIS 包的形式作为数据库作业，由 SQL Server 代理负责运行。

若要运行维护计划，可以通过运行其对应作业的方式来实现，另外，也可以通过直接运行维护计划的方式来实现。例如要运行数据库备份的维护计划 DBbackupPlan，只需要在对象资源管理器中选择该节点，在弹出的快捷菜单中选择“执行”选项即可。

维护计划虽然最终以 SSIS 包的形式，在 SQL Server 代理中作为作业执行，但是维护计划的执行日志却与作业日志有所不同。在对象资源管理器中右击某个维护计划，然后在弹出的快捷菜单中选择“查看历史记录”选项，打开该维护计划的日志文件查看器。维护计划的日志中分别记录了每个任务的执行时间、执行的结果、执行单个任务所花的时间等。

而在作业历史记录中，整个维护计划的子计划将作为一个作业步骤来执行，所以日志中使用一行日志记录了整个子计划中的任务执行情况，不便于查看。

若要修改维护计划的执行周期、执行时间，则可以直接在 SQL Server 代理中修改其对应的计划，当然也可以通过执行直接修改维护计划的操作来修改具体计划。

由于维护计划在数据库作业中是当作一个作业步骤，所以维护计划中的任务不能通过修改作业的方式来修改，必须通过修改维护计划来完成。在对象资源管理器中右击要修改的维护计划，在弹出的快捷菜单中选择“修改”选项，打开该维护计划的设计界面。与 17.5.2 节中配置维护计划的操作相同，修改其中的任务属性等，然后单击“保存”按钮即可。

17.6　小　　结

本章主要讲解如何使用 SQL Server 代理实现数据库管理自动化。SQL Server 代理是一个单独的 Windows 服务，其中定义了作业、计划、警报和操作员 4 个组件。

作业是 SQL Server 代理执行的一系列指定操作。每个作业是由作业步骤组成，一个作业中可以包含多个作业步骤。作业步骤是作业对数据库或服务器执行的操作。计划指定了作业运行的时间。警报是对特定事件的自动响应。操作员中定义的是负责维护一个或多个 SQL Server 实例的个人联系信息。在为操作员维护了电子邮件信息后，当警报发生时便可通过电子邮件的方式通知操作员。在 SQL Server 代理中可以使用维护计划创建所需的任务工作流，通过维护计划来确保优化数据库、定期进行备份并保持与数据库一致。

第 18 章　商 务 智 能

对于现代的企业，无知是最大的威胁，商务智能所要争取的就是充分利用企业在日常经营过程中搜集的大量数据，并将它们转化为信息和知识，以免除企业中的瞎猜行为和无知状态。本章将简要介绍 SQL Server 2012 在商务智能上的功能。

18.1　商务智能简介

商务智能（Business Intelligence，BI）的概念最早于 1996 年由加特纳集团（Gartner Group）提出，加特纳集团将商务智能定义为：

“商务智能描述了一系列的概念和方法，通过应用基于事实的支持系统来辅助商业决策的制定。商务智能技术提供使企业迅速分析数据的技术和方法，包括收集、管理和分析数据，并将这些数据转化为有用的信息，然后分发到企业各处。”商务智能通常被理解为将企业中现有的数据转化为知识，帮助企业做出明智的业务经营决策的工具。

商务智能的关键是从许多来自不同企业运作系统的数据中提取出有用的数据，并进行清理，以保证数据的正确性。然后经过抽取（Extraction）、转换（Transformation）和装载（Load），即 ETL 过程，合并到一个企业级的数据仓库里，从而得到企业数据的一个全局视图。在数据仓库基础上利用合适的查询和分析工具、数据挖掘工具、OLAP 工具等对其进行分析和处理（这时信息变为辅助决策的知识），最后将知识呈现给管理者，为管理者的决策过程提供支持。

商务智能主要从数字上进行管理。具体说来，商务智能可以在以下 4 个方面发挥作用：

- 理解业务。商务智能通过对数据的分析，帮助用户认识是哪些趋势、哪些非正常情况和哪些行为正对业务产生影响。
- 衡量绩效。商务智能可以用来确立对员工的期望，跟踪并管理其绩效。
- 改善关系。商务智能能为客户、员工、供应商、股东和大众提供关于企业及其业务状况的有用信息，从而提高企业的知名度，增强整个信息链的一致性。
- 创造获利机会。掌握各种商务信息的企业可以出售这些信息从而获取利润。但是，企业需要发现信息的买主并找到合适的传递方式。

在商务智能技术应用中，有这样一个很典型的例子——啤酒与尿布。情况是这样的：曾有一段时间，沃尔玛公司在进行数据分析时发现在美国的店面经常有一种现象：每周啤酒和尿布的销量都会有一次同比攀升，但一时搞不清是什么原因。后来，沃尔玛运用商务智能技术发现，购买这两种产品的顾客几乎都是 25 岁到 35 岁，家有婴儿的男性，每次购买时间均在周末。沃尔玛在对相关数据分析后得出，这些人习惯晚上边看球赛，边喝啤酒，

对于要照顾的孩子，为了图省事就用一次性尿布。得到结果后，沃尔玛决定把这两种商品集中摆在一起，结果两种商品的销量都有了显著增加。

SQL Server 2012 中的商务智能分为集成服务、分析服务和报表服务 3 大模块。集成服务平台可以生成高性能数据集成解决方案，其中包括为数据仓库提取、转换和加载（ETL）包。分析服务为商务智能应用程序提供联机分析处理（OLAP）和数据挖掘功能。而报表服务是一个基于 Web 的集创建、管理、分发于一体的报表平台。

18.2 集 成 服 务

集成服务（Microsoft SQL Server 2012 Integration Services，SSIS）是用于生成高性能数据集成和工作流解决方案（包括针对数据仓库的提取、转换和加载 ETL 操作）的平台。本节将通过简单的示例来说明集成服务的使用。

18.2.1 集成服务简介

集成服务中包括了生成并调试包的图形工具和向导；执行 FTP 操作、SQL 语句执行和电子邮件消息传递等工作流功能的任务；用于提取和加载数据的数据源和目标；用于清理、聚合、合并和复制数据的转换；管理服务，即用于管理集成服务包的集成服务；以及用于对集成服务对象模型编程的应用程序接口（API）。

集成服务被称为是 BI 平台的黏合剂，其具有以下几方面的应用：

- 合并来自异类数据存储区的数据。集成服务可以连接到各种各样的数据源，包括单个包中的多个源。包可以使用.NET 和 OLE DB 访问接口连接到关系数据库，使用 ODBC 驱动程序连接到多个早期数据库，还可以连接到平面文件、Excel 文件和 Analysis Services 项目。集成服务可以实现从这些数据源中提取、转换、合并数据，然后将数据加载到一个或多个目标。
- 填充数据仓库和数据集市。数据仓库和数据集市中的数据通常会频繁更新，因此数据加载量通常会很大。使用集成服务可以将数据大容量加载到 SQL Server 中。
- 清除数据和将数据标准化。集成服务包可以使用精确查找或模糊查找来找到引用表中的值，通过将列中的值替换为引用表中的值来清理数据。
- 将商务智能置入数据转换过程。集成服务提供了用于将商务智能置入 SSIS 包的容器、任务和转换。其中的容器组件通过枚举文件或对象和计算表达式来支持重复运行工作流，甚至也可以通过计算数据根据结果来重复运行工作流。
- 使管理功能和数据加载自动化。集成服务包提供了备份和还原数据库、复制 SQL Server 数据库及其包含的对象、复制 SQL Server 对象和加载数据等日常管理功能。对 OLTP 或 OLAP 数据库环境的管理一般包括数据的加载。集成服务包提供了几个使数据大容量加载更加便利的任务，另外，还可以使用 SQL Server 代理作业来安排 SSIS 包。

18.2.2　使用导入导出向导转换数据

导入导出向导用于帮助用户轻松快捷地创建数据的导入导出工作。利用导入导出向导可以使用 SQL Server、Access、Oracle 等数据库，以及 Excel、平面文件等格式文件相互之间的数据转换工作，当然也可以实现从 SQL Server 到 SQL Server 这样的同种数据库的数据导入导出工作。

在开始菜单中的 SQL Server 2012 文件夹下提供了“导入导出数据（32 位）”选项，另外，也可以通过在“开始”中运行 dtswizard 命令启动导入导出向导。除了这两种方式外，还可以在 SSMS 中右击某个数据库，在弹出的快捷菜单中选择“任务”菜单下的“导入数据”或者“导出数据”选项。例如有一个客户提供的 Excel 格式的预算表，现在需要将 Excel 中的数据导入到数据库中，则对应的操作如下。

（1）在“开始”菜单中选择“导入导出数据（32 位）”选项，启动导入导出向导。

（2）单击“下一步”按钮，进入选择数据源界面，如图 18.1 所示。

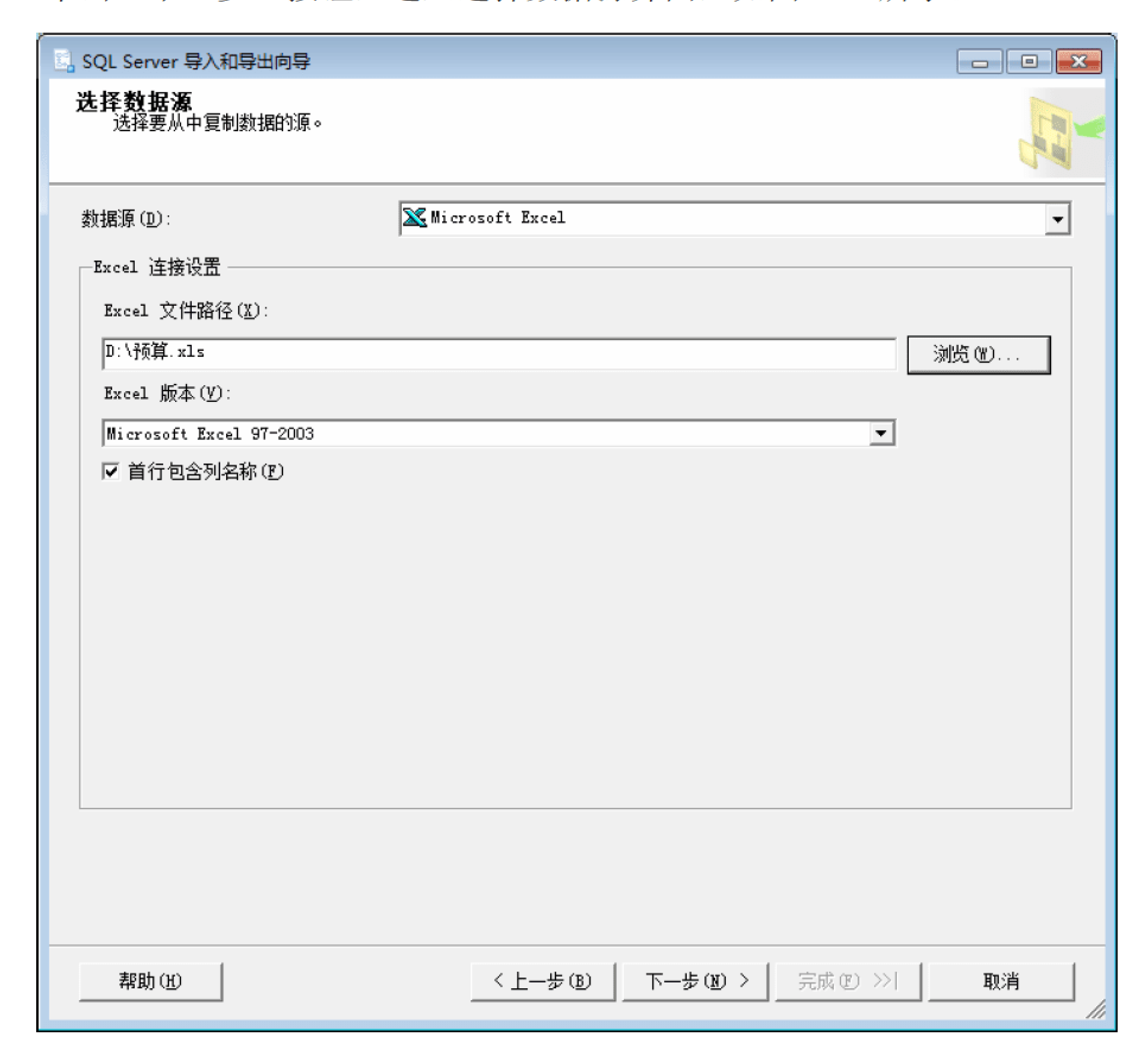


图 18.1　选择数据源

在“数据源”下拉列表框中提供了当前环境支持的所有数据源类型，这里的选项取决于当前环境的驱动。如果安装了 Sybase 数据库的驱动，则可以把 Sybase 数据库作为数据源。这里由于是要将 Excel 数据导入到 SQL Server 中，所以选择 Microsoft Excel 选项。在

“Excel 文件路径”文本框中输入要导入的 Excel 文件的完整路径。在“Excel 版本”下拉列表框中提供了从 Excel 3.0 到 Excel 2007 的支持，这里由于是使用 xls 格式的 Excel 2003 文件，所以选择 Microsoft Excel 97-2003 选项。“首行包含列名称”复选框表示在 Excel 中第一行是否表示为该列的列名。

（3）单击“下一步”按钮，进入选择目标界面，如图 18.2 所示。

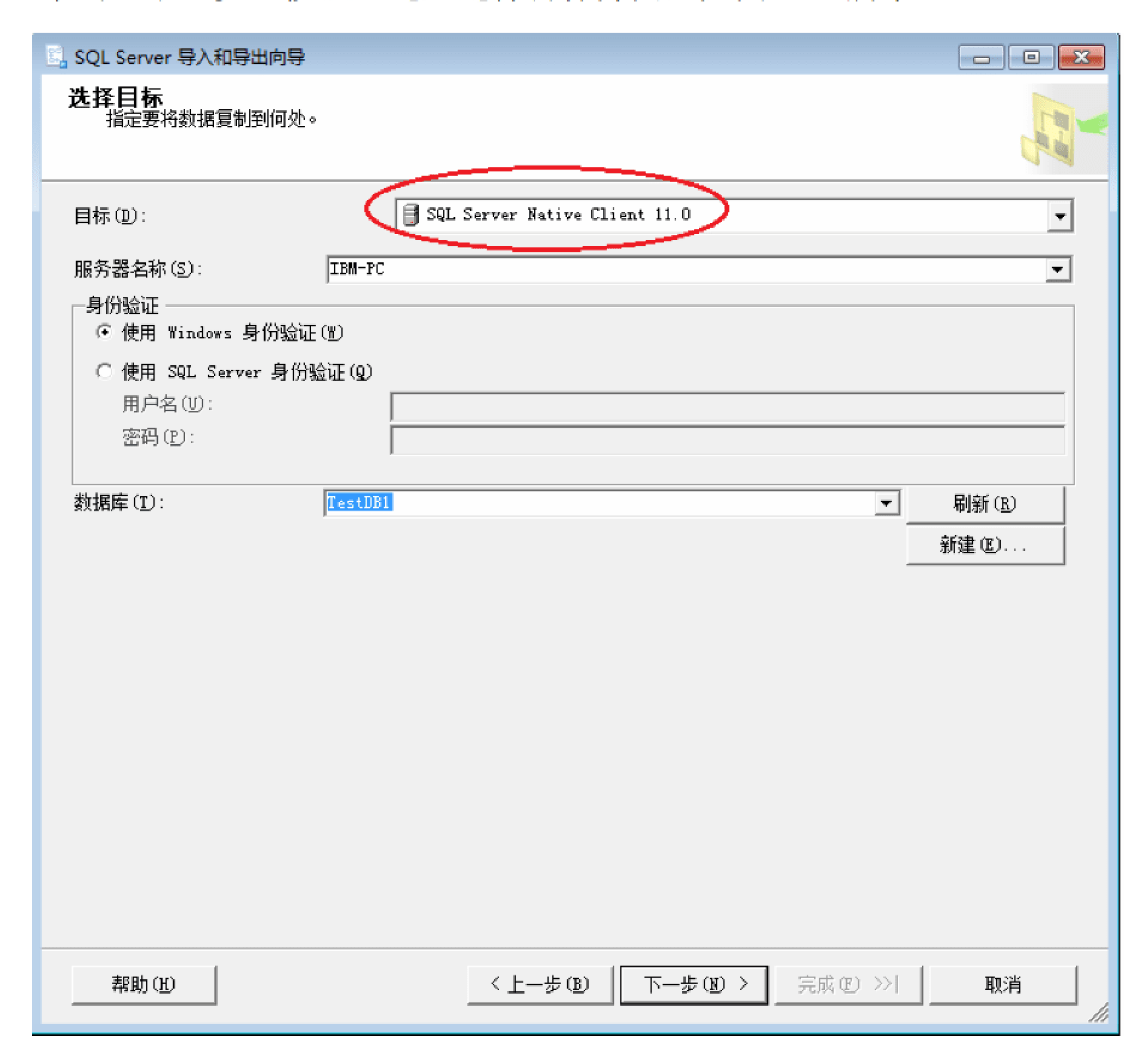


图 18.2　选择目标界面

在其中的“目标”下拉列表框中选择“SQL Server Native Client 11.0”选项，表示要导入到 SQL Server 2012 数据库中。接下来输入数据库所在的服务器名称、身份认证和要导入的数据库。如果是导入全新的数据库，则单击“新建”按钮来创建和导入数据库。

（4）单击“下一步”按钮进入指定表复制或查询界面，这里需要复制的是一个表，所以选择“复制一个或多个表或视图的数据”选项。

（5）单击“下一步”按钮进入选择源表和源视图界面，该界面列出了 Excel 源中的所有 Sheet，如图 18.3 所示。

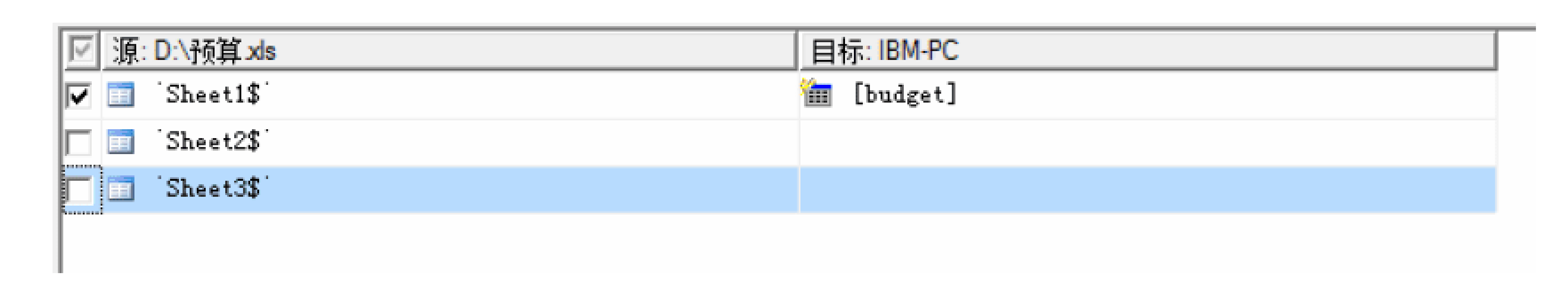


图 18.3　选择源表

这里要导入的数据在“预算”Sheet 中，所以选中“Sheet1$”选项左边的复选框，目标列则选择要导入表的表名。如果要导入到一个新表中，则直接输入目标表名即可，这里由于是全新的导入，所以输入新表名 budget。

（6）单击“编辑映射”按钮，进入“列映射”对话框，如图 18.4 所示。映射表中“源”列为 Excel 中分析出的列名，“目标”列中为导入到 SQL Server 后的列名，该列名可以修改，也可以忽略。“类型”列则是导入到 SQL Server 后列的数据类型。其余的列还可以设置导入后的列是否为空，以及数据长度、精度和小数位数等。

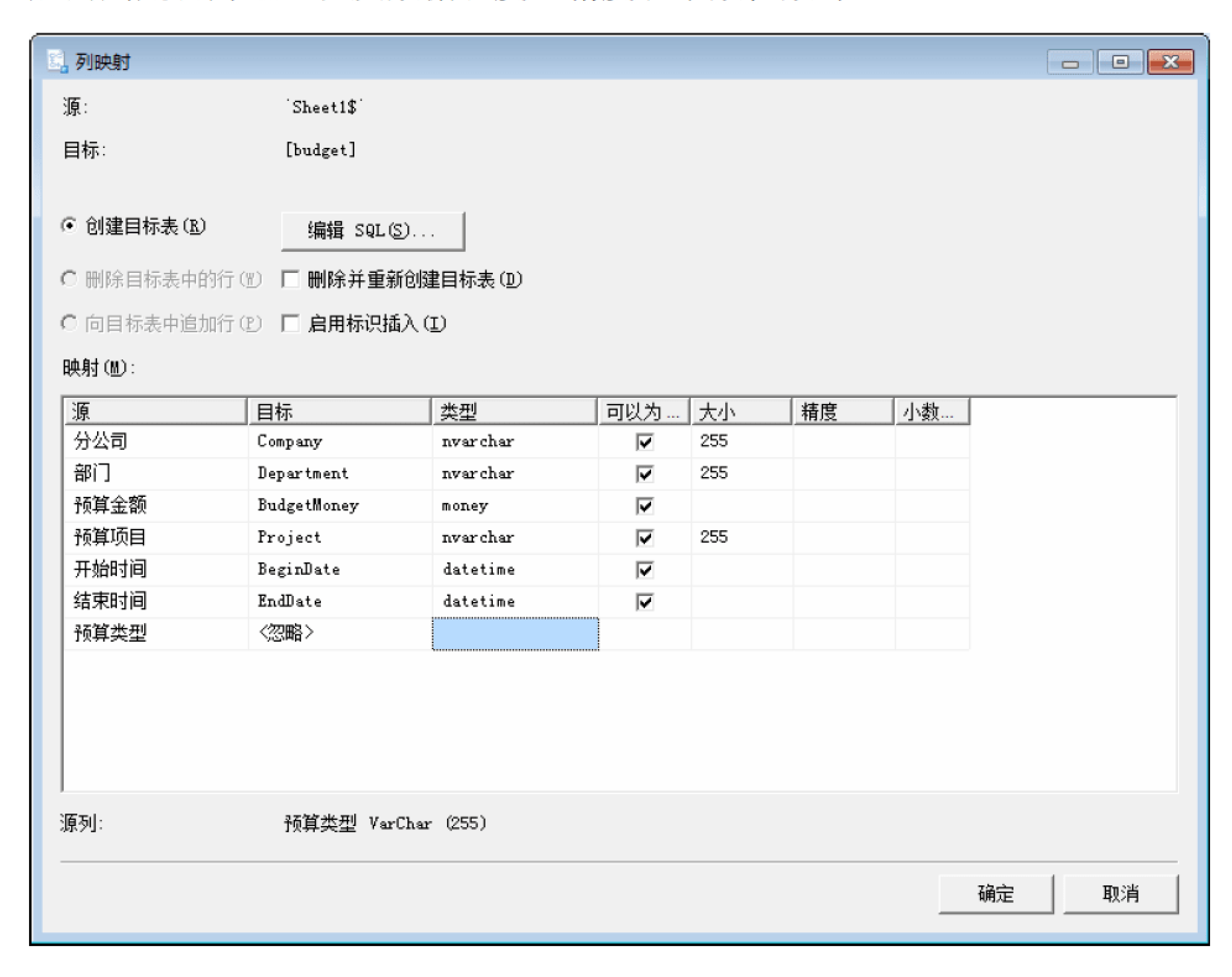


图 18.4 “列映射”对话框

（7）修改好目标列的属性后单击“确定”按钮，回到向导界面，然后单击“下一步”按钮进入保存并运行包界面，如图 18.5 所示。

立即运行(R)
保存 SSIS 包(S)
SQL Server(Q)
文件系统(F)
包保护级别(L):
使用用户密钥加密敏感数据
密码(P):
重新键入密码(R):

图 18.5 保存并运行包

通过导入导出向导，系统将生成一个 SSIS 包，此处可以将 SSIS 包保存到文件系统或者 SQL Server 中，这样在下次导入导出同样的数据时，就不需要重复配置导入导出了，如果只运行一次当然也没有必要保存 SSIS 包。这里选择保存 SSIS 包到文件系统以便下次修改和使用。选中“立即运行”复选框表示在向导完成时将执行导入工作。“包保护级别”下拉列表框中提供了多个用户密码的保护策略包含在导入导出时，连接数据库的用户认证信息。

（8）单击“下一步”按钮，由于选择了保存 SSIS 包，所以进入保存 SSIS 包的设置界面。输入 SSIS 包的名称和说明，选择该 SSIS 包保存的路径。

（9）单击“完成”按钮系统将开始执行数据导入工作，在执行完成后将报告导入成功的数据行数。如果设置了保存 SSIS 包到文件系统，则可以在设置的路径下看到生成的 dtsx 文件。至此 Excel 中的数据已经复制到 SQL Server 数据库中。

18.2.3　Excel 数据的导入导出

假设现在有个财务数据库 Finance，其中存在一个预算表 Budget，记录了各个分公司各部门的每个会计科目的预算情况，现在业务人员用 Excel 制作好了新的预算表，需要将 Excel 中的数据导入到 Finance 数据库中。由于提供的 Excel 是按照标准预算模板填写，所以可以使用 SSIS 将数据转存到数据库中，具体操作如下。

（1）打开 VS 2010，选择“新建项目”选项，在弹出的新建项目对话框中选择商业智能项目中的“Integration Services 项目”选项，如图 18.6 所示。

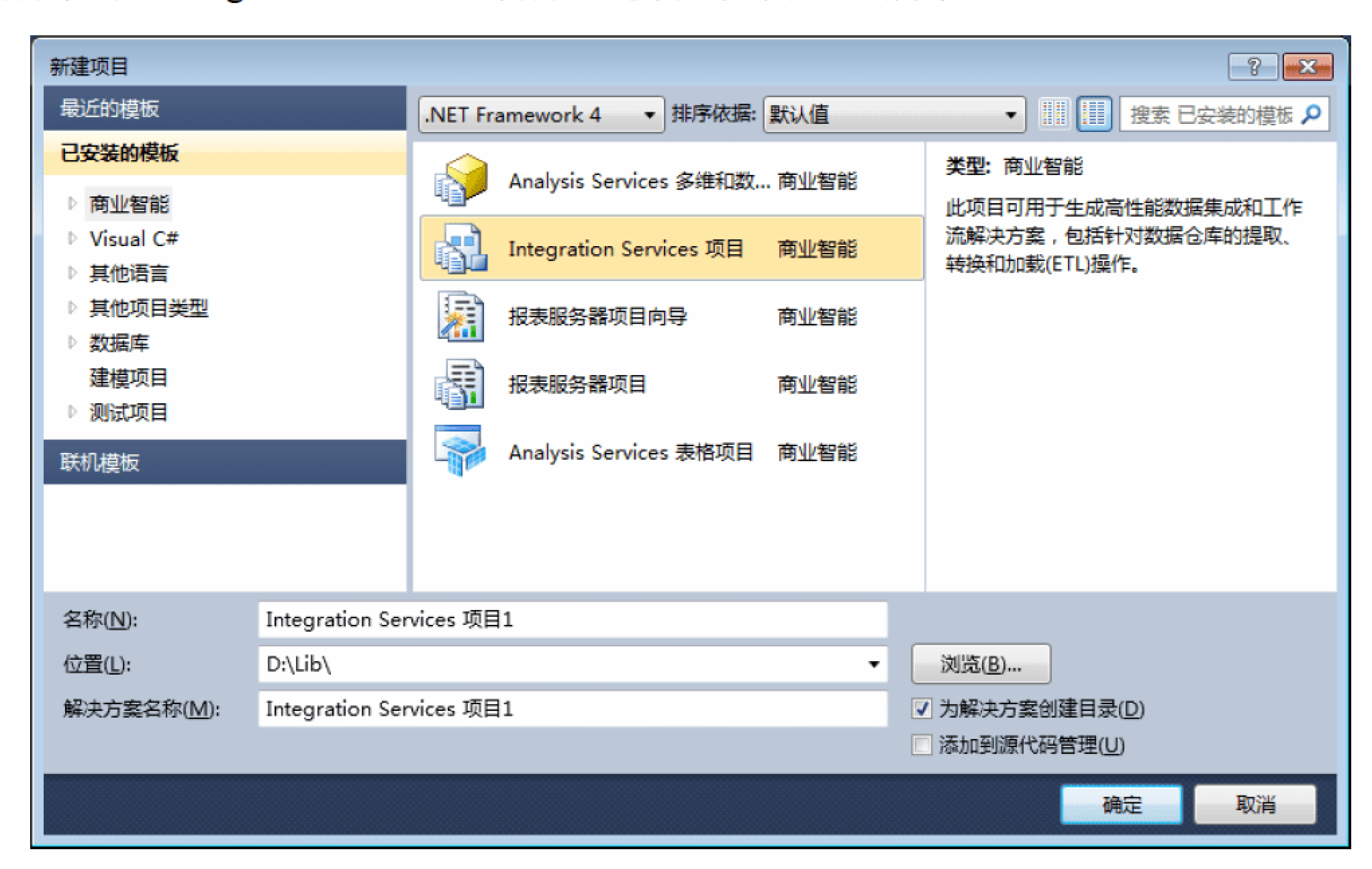


图 18.6　新建集成服务项目

（2）在“名称”文本框中输入项目名称 SSIS1，然后单击“确定”按钮，进入集成服务设计工作界面，如图 18.7 所示。

图 18.7 左边的工具箱提供了要执行 SSIS 任务所用到的所有工具，中间区域为任务的

设计区域，右边是该项目的各个任务和属性窗口。

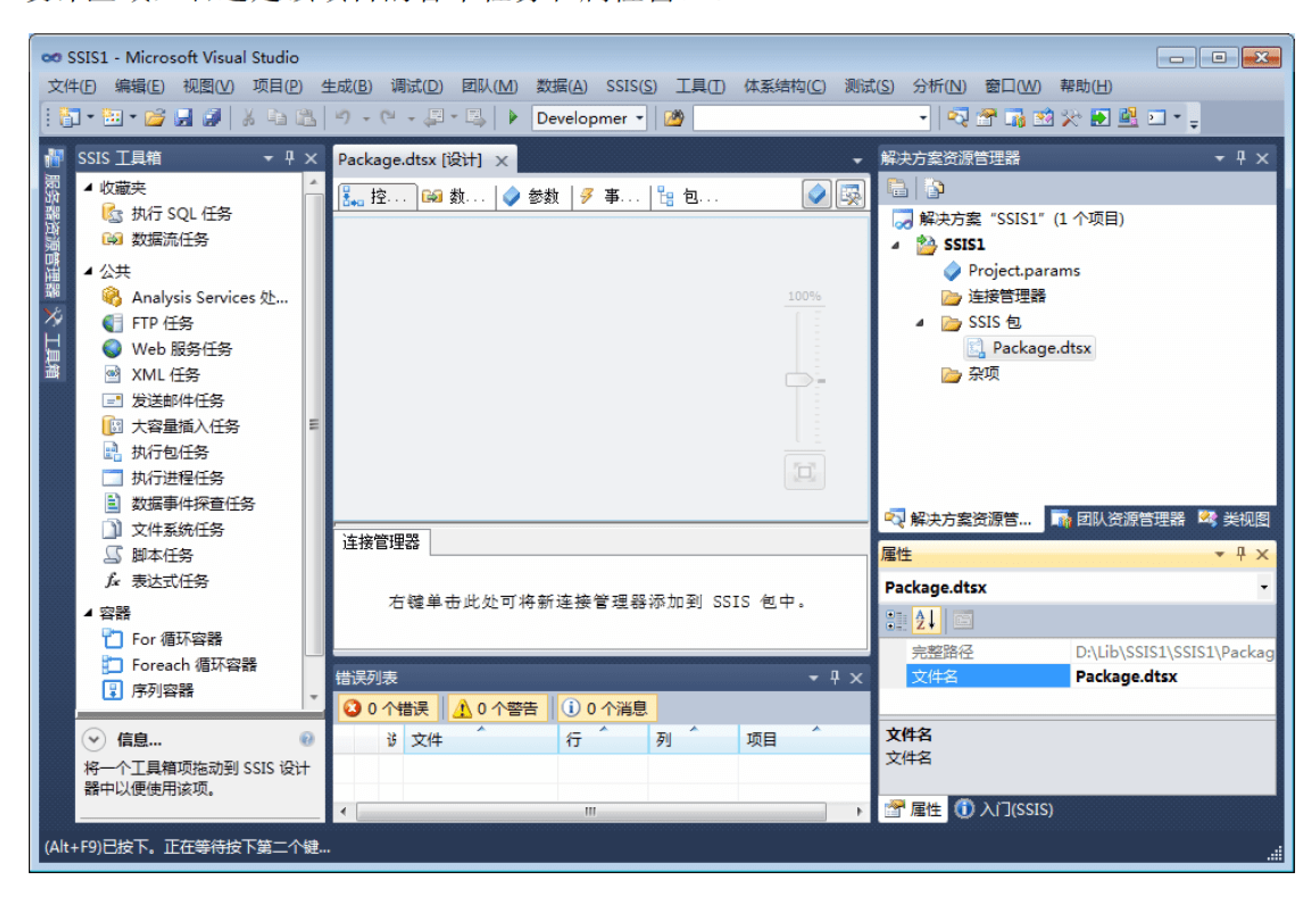


图 18.7　集成服务设计界面

（3）SSIS 包中的控制流用不同类型的控制流元素构造而成：容器、任务和优先约束。容器提供包中的结构并给任务提供服务，任务在包中提供功能，优先约束将容器和任务连接成一个控制流。在“控制流”面板中，从工具箱中拖曳一个“数据流任务”到其中。数据流任务封装数据流引擎，该引擎在源和目标之间移动数据，使用户可以在移动数据时转换、清除和修改数据。将数据流任务添加到包控制流，使包可以提取、转换和加载数据。

（4）在主设计面板中，切换到“数据流”选项卡，工具箱中的组件也自动切换为了数据流设计的组件。数据流设计组件分为：数据流源、数据流转换和数据流目标 3 大类。一个完整的数据流就是从数据流源开始，经过零到多个数据流转换操作，最终在数据流目标中结束。从工具箱中拖曳一个 Excel 源和一个 OLE DB 目标组件到数据流面板中。

（5）拖曳 Excel 源下的蓝色箭头到 OLE DB 目标上，表示数据从 Excel 源传输到 OLE DB 目标中。

说明： 绿色箭头表示成功处理结束的数据流，红色箭头则表示处理过程中出现了错误的数据。

（6）右击“Excel 源”组件，在弹出的快捷菜单中选择“编辑”选项，弹出“Excel 源编辑器”对话框，如图 18.8 所示。单击“新建”按钮，将 Excel 文件路径添加到 OLE DB 连接管理器中。然后在“Excel 工作表的名称”下拉列表框中选择预算工作表。

（7）选择“列”选项页，切换到列输出配置界面，如图 18.9 所示。

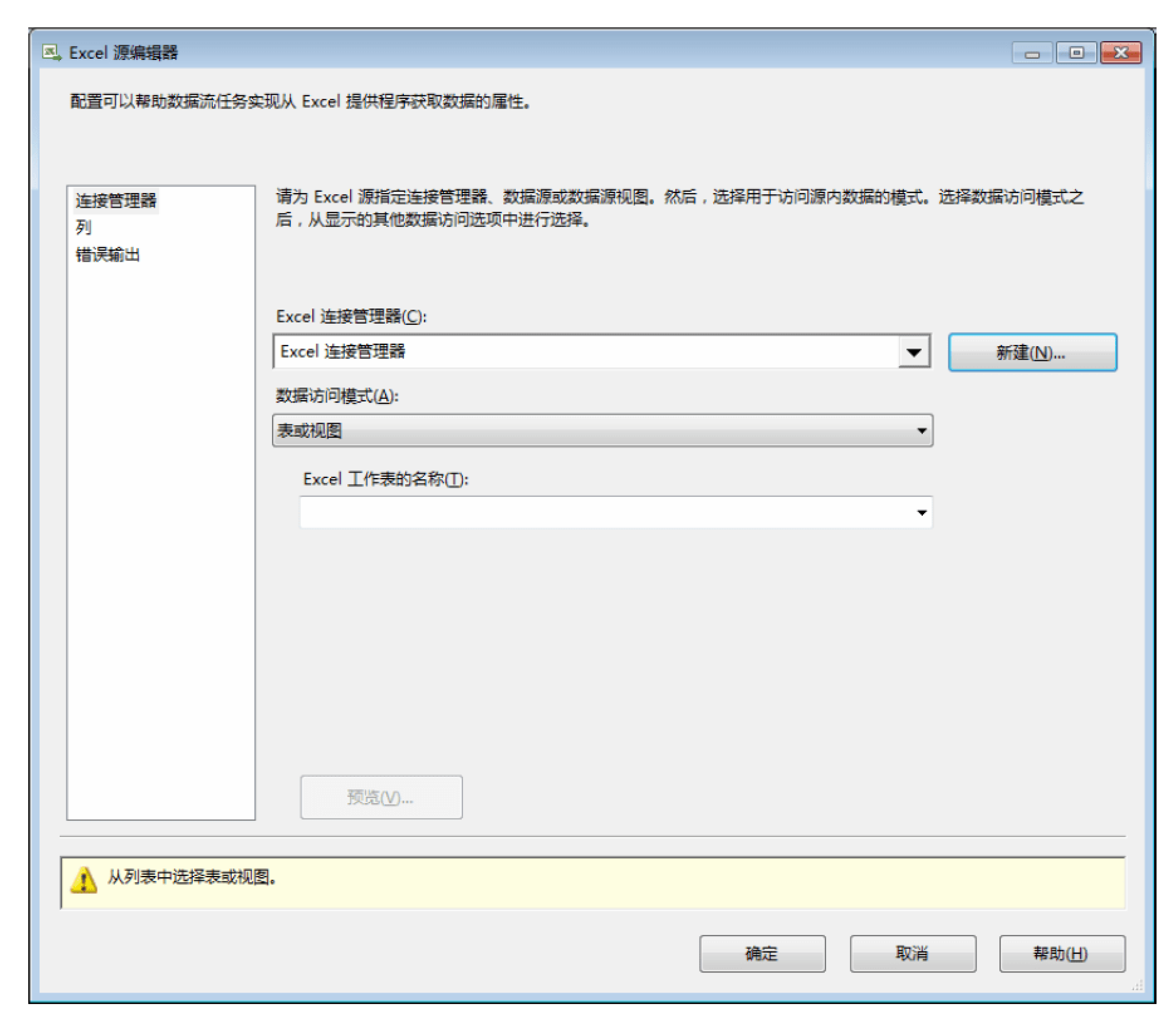


图 18.8　“Excel 源编辑器”对话框

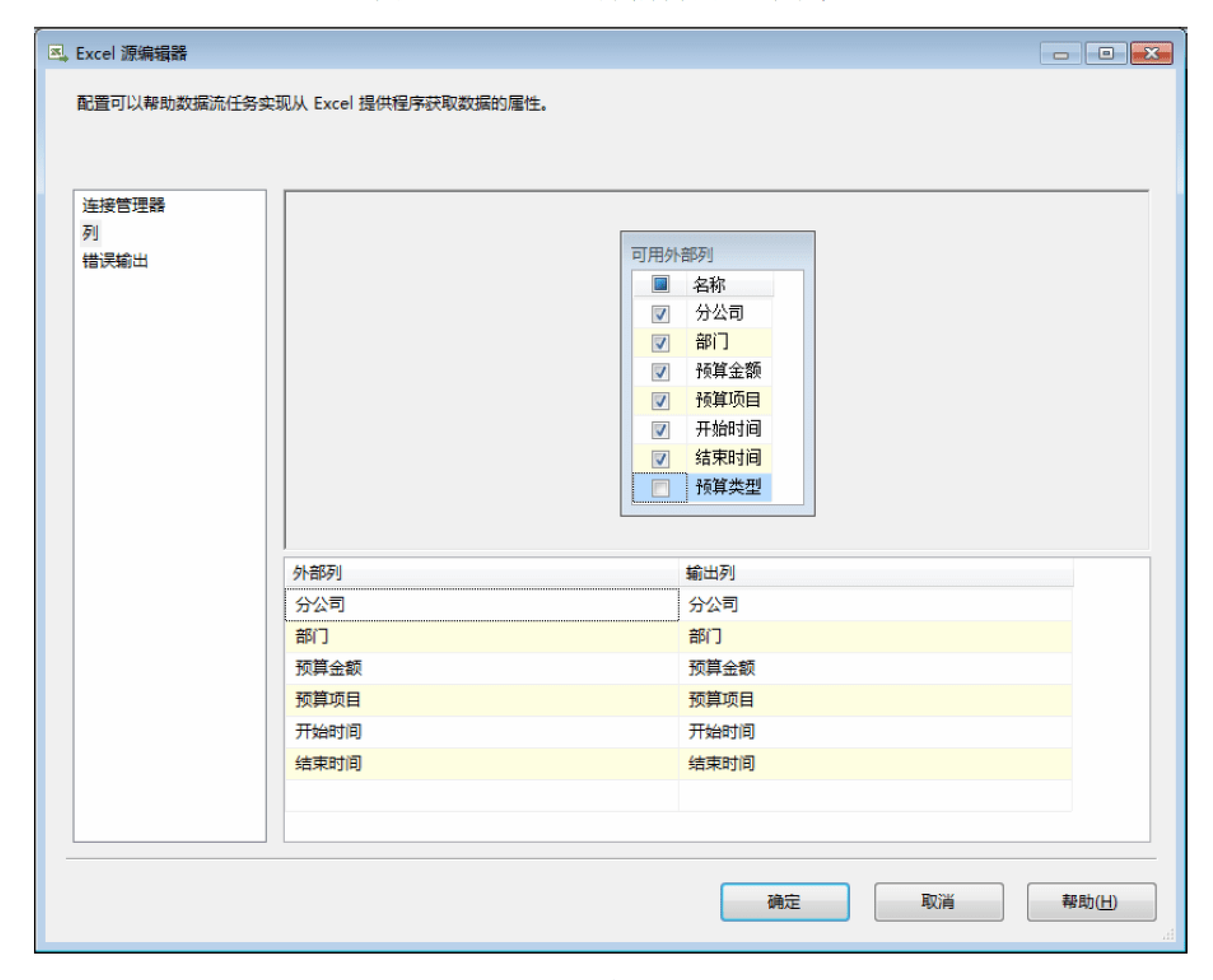


图 18.9　列输出配置界面

这里列出了从 Excel 数据中分析出来的列名，如果不用导入的列，则可以去掉该列左边的复选框。

（8）单击“确定”按钮，回到数据流设计界面，用同样的方法为 OLE DB 目标设置连接的 SQL Server 数据库和表，然后选择“映射”选项页，切换到输入列与输出列的映射设置界面，如图 18.10 所示。

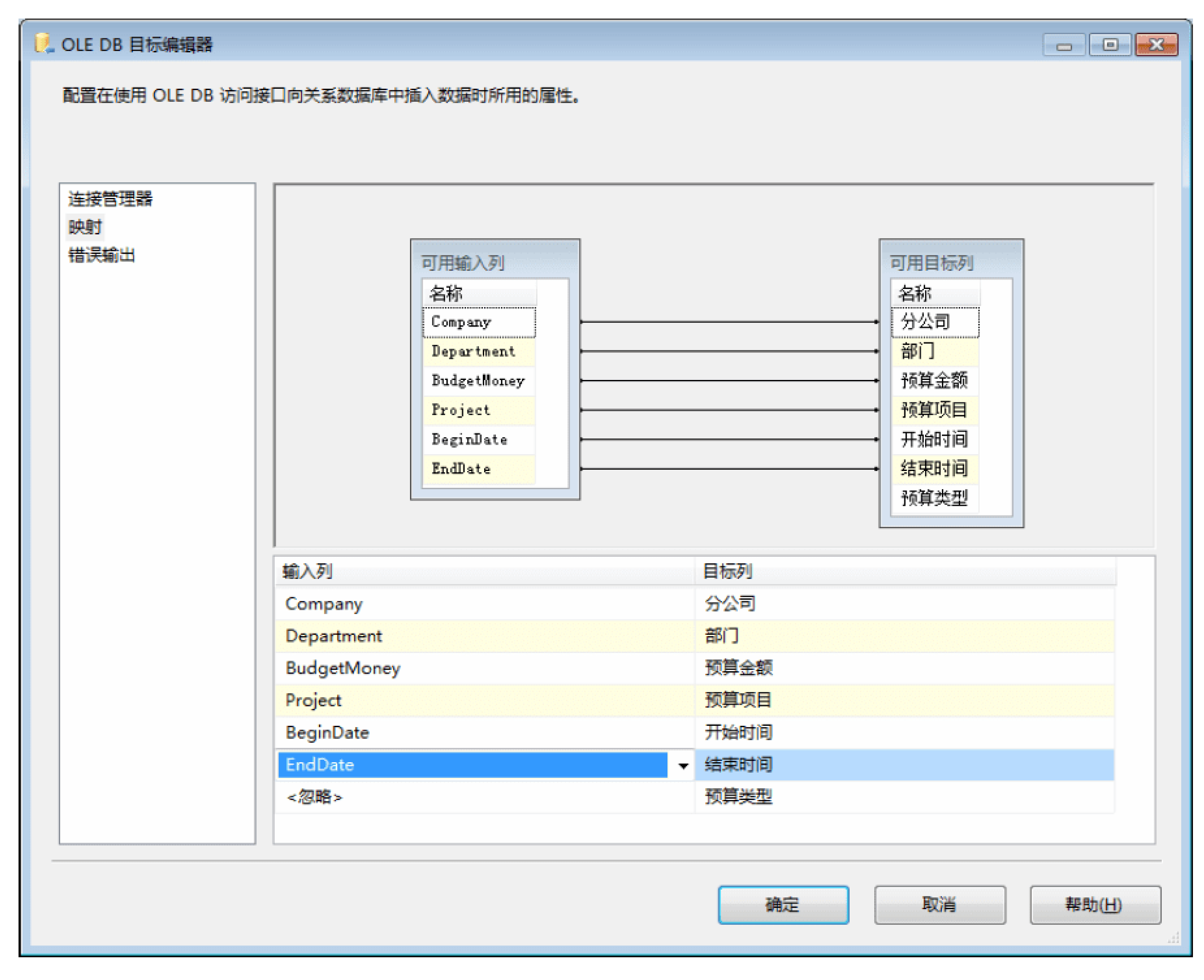


图 18.10　列映射设置

可以在图 18.10 中的列表中通过选择的方式来设置每个输入列和目标列的映射，也可以在上面拖曳可用输入列到可用目标列上，从而实现两个列之间的映射。

（9）单击“确定”按钮回到数据流设计面板，整个 Excel 数据导入的 SSIS 包已经设计完成。单击工具栏的“调试”按钮或者直接使用快捷键 F5，系统将保存该 SSIS 包并运行。运行成功后将提示导入的数据行数，由于目前引用的 Excel 表中没有数据，因此导入行数是 0，如图 18.11 所示。

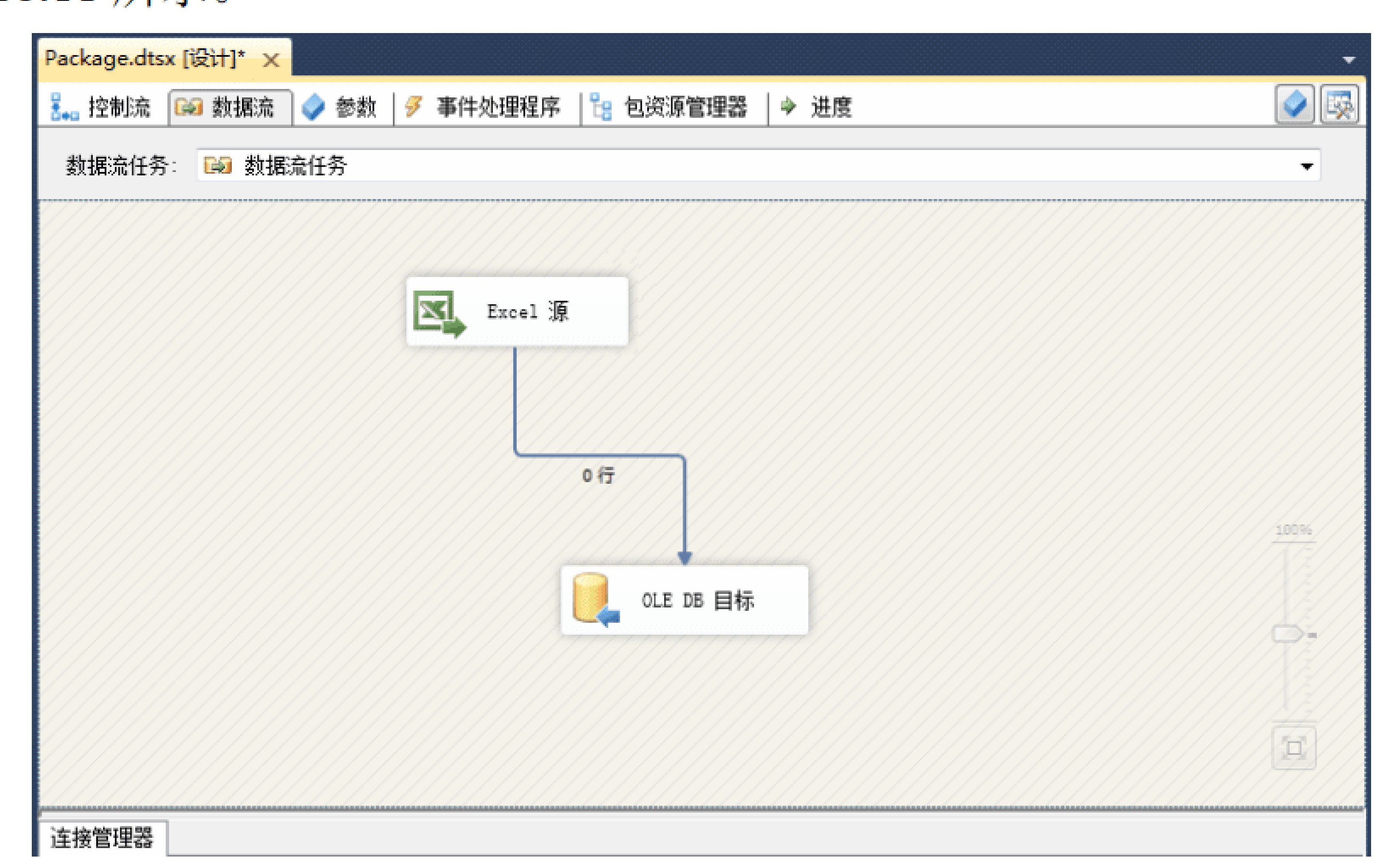


图 18.11　运行 SSIS 包的结果

现在打开数据库，如果 Excel 中有数据的话，就可以看到 Excel 中的数据被成功导入到 SQL Server 中。

18.2.4　数据查找

前面只是将 Excel 中的数据原封不动地转移到 SQL Server 中，但是在实际应用中，往往用户提交的Excel 数据是字符串形式的内容，而数据库中则以内容对应的ID的形式保存。例如导入的公司名称、部门名称等，在 Budget 表中实际保存的是 CompanyID 和 DepartmentID。

若要实现将公司名称转换为公司 ID 进行保存，则需要使用数据流组件中的查找功能。查找转换通过连接输入列中的数据和引用数据集中的列来执行查找。可以使用该查找在基于通用列的值的相关表中访问其他信息。

引用数据集可以是缓存文件、现有的表或视图、新表或 SQL 查询的结果。查找转换使用 OLE DB 连接管理器或缓存连接管理器连接到引用数据集。可以用以下方式来配置查找转换：

- 选择要使用的连接管理器。如果要连接到数据库，选择 OLE DB 连接管理器。如果要连接到缓存文件，选择缓存连接管理器。
- 指定包含引用数据集的表或视图。
- 通过指定 SQL 语句生成引用数据集。
- 指定输入和引用数据集间的连接。
- 将引用数据集的列添加到查找转换输出中。
- 配置缓存选项。

查找转换首先在转换输入的值和引用数据集的值之间执行同等连接（同等连接意味着转换输入中的每一行至少要与引用数据集中的一行匹配）。如果无法实现同等连接，则查找转换会执行下列操作之一。

- 如果引用数据集中没有匹配项，则不会发生连接。默认情况下，查找转换将没有匹配项的行视为错误。但是，可以将查找转换配置为将这些行重定向到无匹配输出。
- 如果引用表中有多个匹配项，则查找转换只返回查找查询返回的第一个匹配项。如果发现多个匹配项，则仅当转换被配置为将所有引用数据集加载到缓存中时，查找转换才生成错误或警告。在这种情况下，如果查找转换在填充缓存时检测到多个匹配项，则该查找转换将生成警告。

通常，将来自引用数据集的值添加到转换输出中。

注意： 查找转换执行的查找区分大小写。因此，为了避免因数据中大小写不同而导致查找失败，首先使用字符映射转换将数据转换为大写或小写。然后在生成引用表的 SQL 语句中包含 UPPER()或 LOWER()函数。

例如，现在有公司表 Company，在进行 Excel 导入时，需要使用分公司的名字在该表中找出其对应的 ID，然后将公司 ID 输出给 OLE DB 目标，则对应的操作如下。

（1）使用 VS 打开前面创建的 SSIS 项目，切换到“数据流”设计面板。

（2）从工具箱中拖曳“查找”组件到设计面板中，然后删除原来 Excel 源到 OLE DB 目标之间的绿色箭头连接，重新将 Excel 源的蓝色箭头拖曳到查找组件上。

（3）右击“查找”组件，在弹出的快捷菜单中选择“编辑”选项，系统将打开查找转换编辑器窗口。

（4）“常规”选项卡中主要是设置缓存模式、连接类型和如何处理无匹配的项，这里都保持默认值，选择“连接”选项卡，进入查找连接设置界面，将要查找的数据库和表 Company 添加到连接设置中。

（5）选择“列”选项卡，切换到列设置界面如图 18.12 所示。可用输入列就是从 Excel 数据源传入的列，将分公司拖曳到可用查找列的 Company 上，系统将为这两列建立联系，表示通过分公司列在 Company 表中查找 Company 列。在可选查找列中选中 CompanyID 选项，下面将列出查找列为 CompanyID，查找操作是作为新列添加，输出别名为 CompanyID。

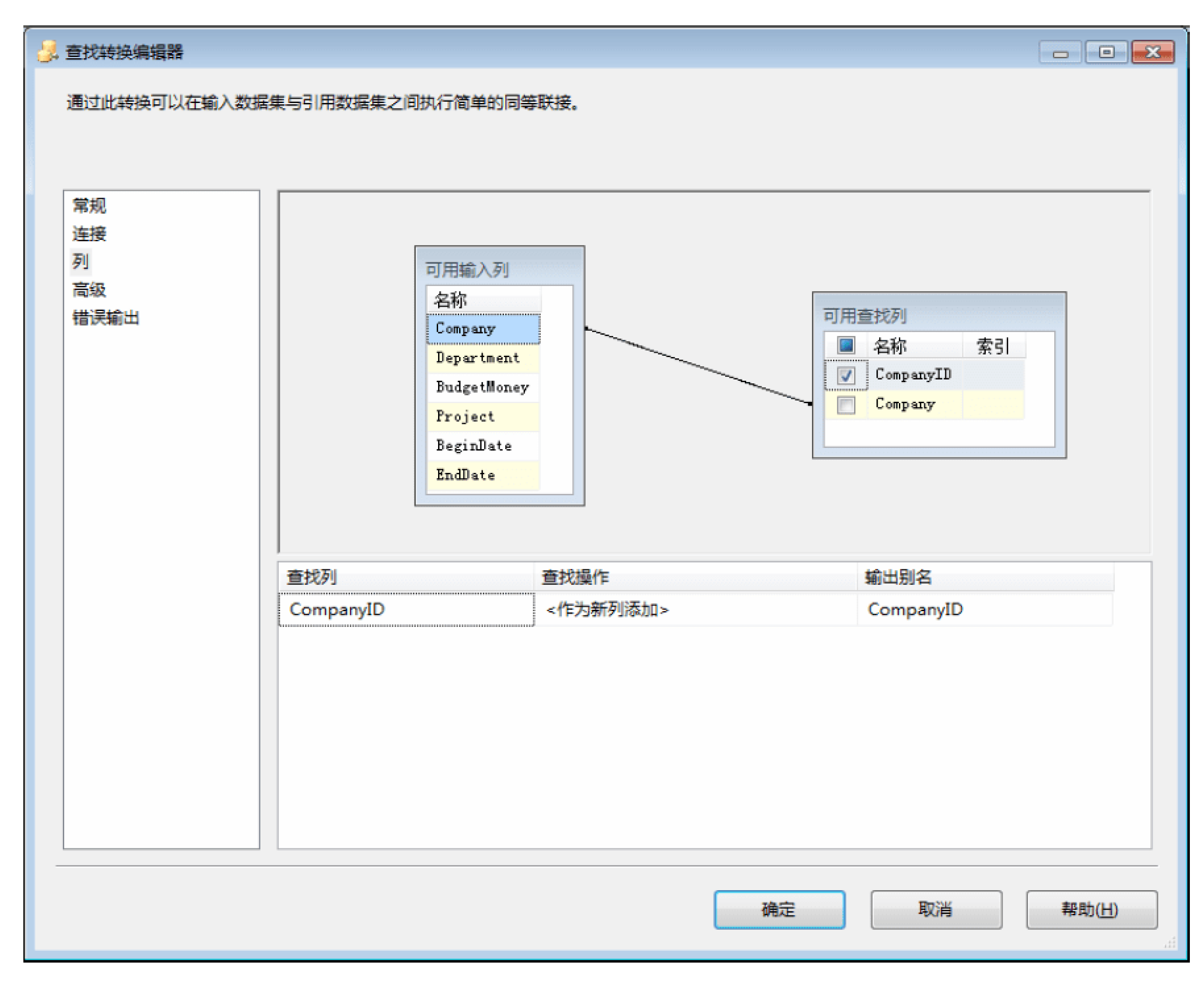


图 18.12　查找列设置

（6）单击“确定”按钮回到数据流设计面板，将查找组件的蓝色箭头拖曳到 OLE DB 目标上，弹出“选择输入输出”对话框，如图 18.13 所示。

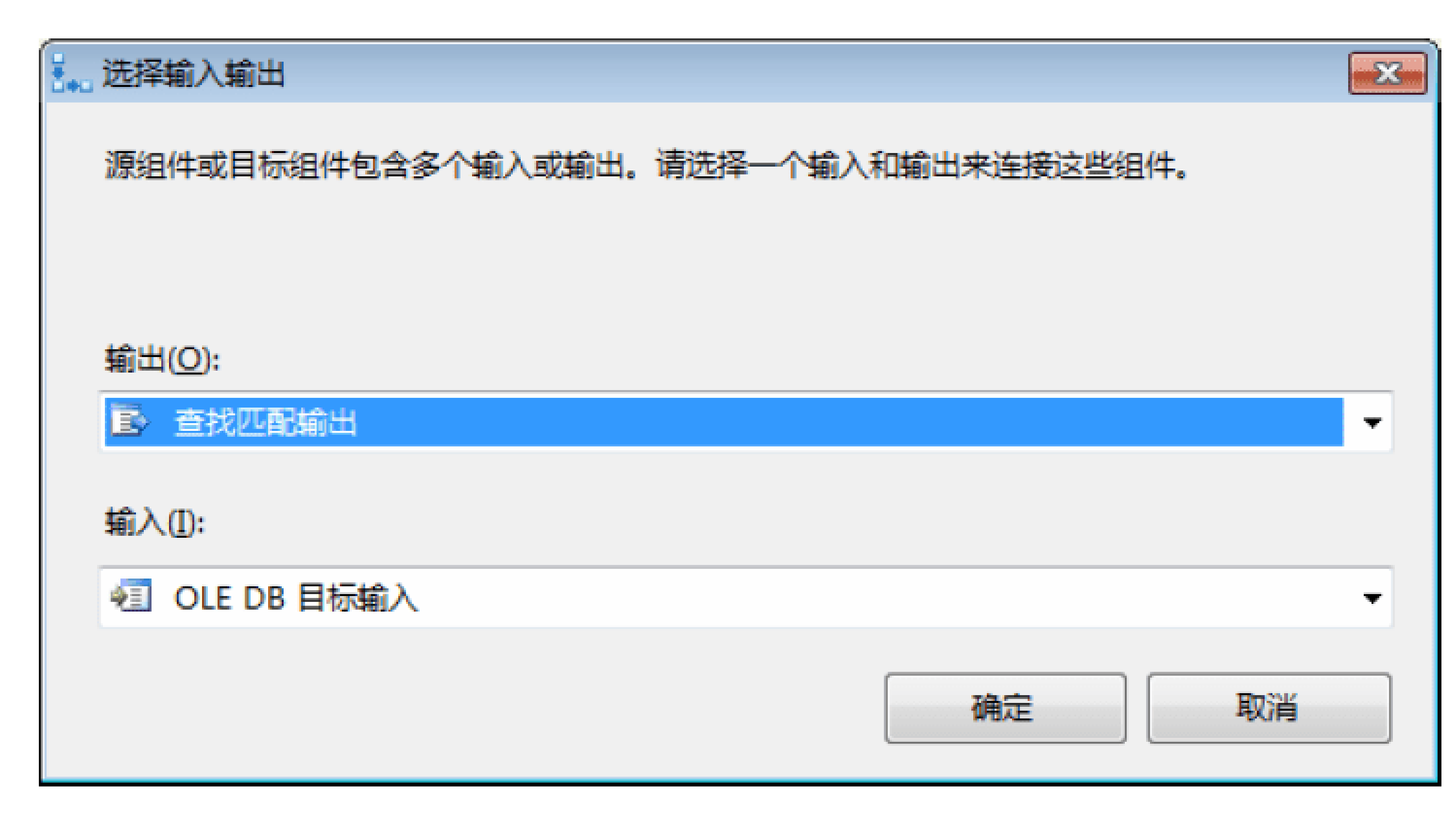


图 18.13　“选择输入输出”对话框

在“输出”下拉列表框中选择“查找匹配输出”选项作为 OLE DB 目标的输入，表示将查找的公司 ID 的数据输入到 OLE DB 目标中。

（7）新建具有 CompanyID 列的 Budget 表，在 OLE DB 目标中将查找的结果与 SQL Server 表进行映射。

至此，数据查找的工作已经完成，按下 F5 键运行该 SSIS 包，可以看到所有的数据已经被正确查找并最终输出到 SQL Server 数据库中，如图 18.14 所示。

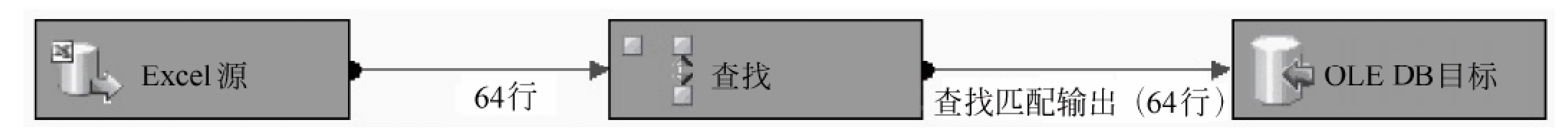


图 18.14　查找转换结果

18.2.5　数据处理

除了查找组件外，SSIS 还提供了很多个数据处理的组件，例如进行数据类型转换的数据转换组件，用于将一个列复制成多个列的复制列组件，对原有列做简单的数据处理后生成新列的派生列组件等。

从 Excel 数据源输出的字符串默认长度是 255，但是在数据库中一般也用不了那么长，例如将部门名保存到数据库中，从 Excel 中获得的部门列的数据类型是 nvarchar(255)，但是在 SQL Server 现有表中，设计 Budget 表的 Department 列的数据类型是 nvarchar(20)，则在使用 SSIS 导入数据时将会发生异常。

对于这种数据类型不匹配的列，需要使用“数据转换”组件。在数据流面板中将数据转换组件插入到查找与 OLE DB 目标之间。然后双击该组件，弹出“数据转换编辑器”对话框，将部门列设置为输入列，输出别名为 DepartmentName，数据类型为 Unicode 字符串，长度修改为 20，如图 18.15 所示。

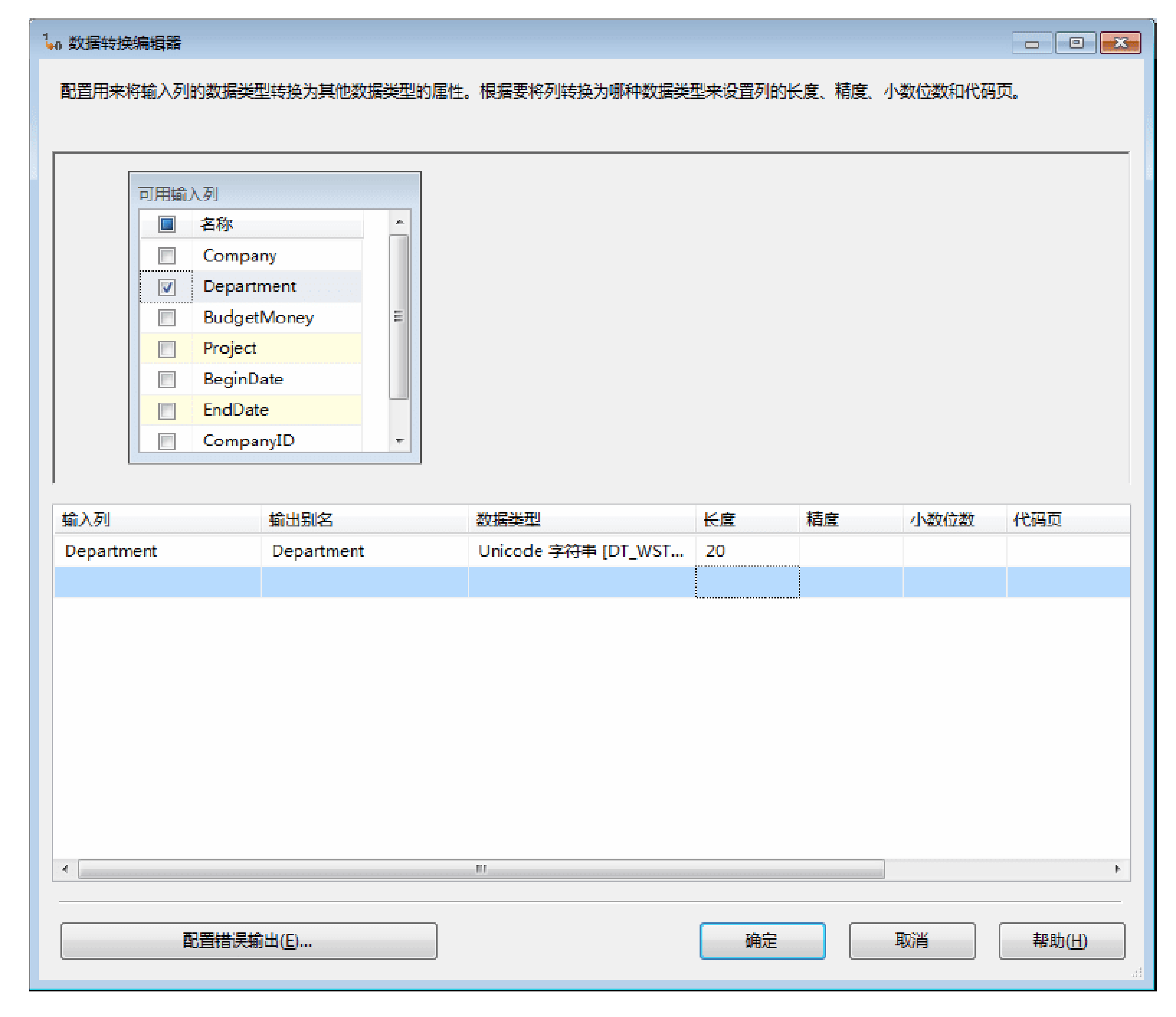


图 18.15　“数据转换编辑器”对话框

如果 SQL Server 数据库中 Budget 表还有其他列，例如有个状态列 Status，而该列不允许为空，所有导入的数据其 Status 列都是 1。另外还有一个 Year 列，表示预算开始的年度，该列可以从开始时间列中计算得出，这些情况下就需要使用派生列组件。

派生列转换使用表达式更新现有值或向新列中添加值。对于前面的情况，在 SSIS 包的数据流设计面板中添加“派生列”组件，将该组件添加到整个数据流中的数据转换和 OLE DB 目标之间。双击该组件，弹出“派生列转换编辑器”对话框，如图 18.16 所示。

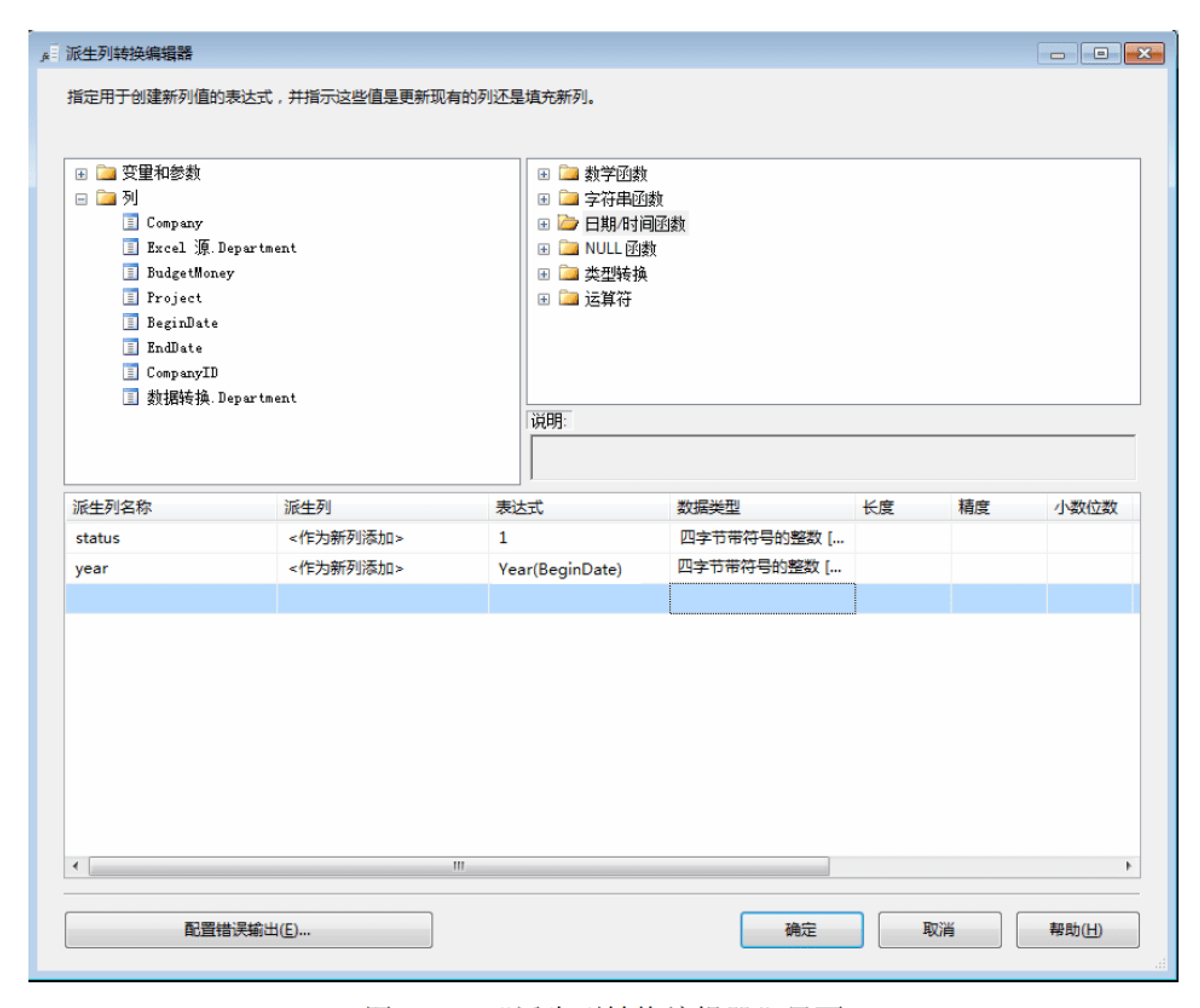


图 18.16 “派生列转换编辑器”界面

对话框左上角列出了输入的列，右上角则给出帮助用户查看的各种数据函数。要添加 Status 列，在下面派生列列表中输入派生列名称 Status，作为新列添加，表达式则使用要求的值 1，数据类型等将会根据表达式自动给出，一般不用修改。同样的方法添加派生列 Year，其表达式为 Year([开始时间])。设置好派生列后接下来就是修改 OLE DB 目标，使对应的列进行映射。

现在 Excel 数据在经过查找、数据转换、派生列等处理后，才最终写入到 SQL Server 中，按下 F5 键运行该 SSIS 包，可以看到运行的结果，如图 18.17 所示。

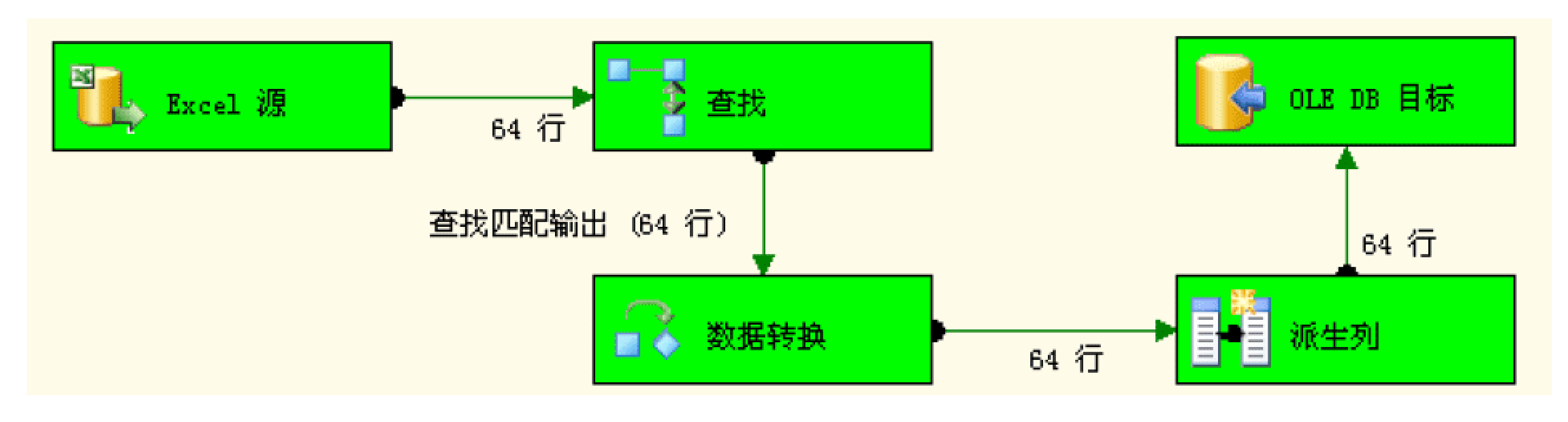


图 18.17　运行数据处理的 SSIS 包

注意： 很多情况下数据流处理组件并没有使用其他组件生成的数据列，各个组件之间并没有明显的先后顺序，所以是先查找还是先进行数据转换，或者是先生成派生列，对整个数据流并不会有任何影响。

由于进行数据处理的 SSIS 组件很多，笔者在此不逐一介绍，读者若需要了解每个组件的功能和用法，可以查看联机丛书。

18.2.6　异常处理

之前讲解的内容都是在数据良好的情况下做的数据处理，也就是说从 Excel 中读取出来的数据都是理想而符合规范的。但是在实际生活中由于限制不够严格或者用户操作失误等原因导致导入的数据并不完全符合规范，这种情况下进入数据查找、数据转换等操作可能失败，从而导致整个导入操作失败。对于这些发生异常的情况则需要进行异常处理。

SSIS 中使用红色箭头来表示异常的数据流。在运行 SSIS 包时，如果某个组件运行失败，也是使用红色背景来表示。同样以这个预算 Excel 表格为例，人为地修改其中的 2 行数据，使得其分公司名在数据库 Company 表中并不存在，那么在做数据导入时，SSIS 包运行到查找组件时将引发异常，从而导致整个包运行失败。失败的 SSIS 包如图 18.18 所示。

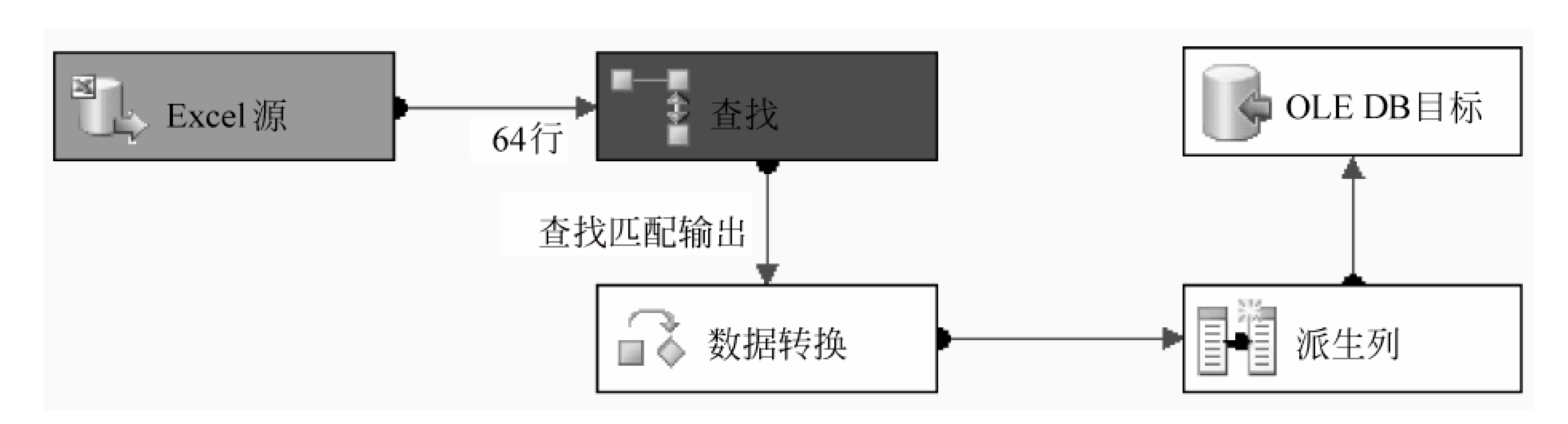


图 18.18　查找失败的 SSIS 包

切换到“进度”面板，可以看到以下提示信息：

```
[查找 [65]] 错误：在查找期间行没有生成任何匹配项。
[查找 [65]] 错误：SSIS 错误代码 DTS_E_INDUCEDTRANSFORMFAILUREONERROR。“组件“查找”(65)”失败，错误代码为 0xC020901E，而且针对“输出“查找匹配输出”(67)”的错误行处理设置指定一旦出错就失败。在指定组件的指定对象上出错。可能在此之前已经发出错误消息，提供了有关失败的详细信息。
```

从错误提示中可以得出，查找组件在运行时有的数据并没有任何匹配项，所以导致了该错误。对于这种错误的数据，这里并不知道是哪一行，哪个错误，这种情况下就需要将错误数据输出，便于用户排查。这里采用将所有错误的数据输出到一个文本文件中的方式，来排查错误数据，而正确匹配的数据正常导入数据库中，在 SSIS 中的操作如下。

（1）在数据流设计面板中将“平面文件目标”组件添加到其中，用于记录错误的数据。

（2）将“查找”组件的红色箭头拖曳到“平面文件目标”组件中，弹出“配置错误输出”对话框，如图 18.19 所示。

（3）将错误列的行修改为重定向行，表示错误的数据都重定向到错误处理流程中，然后单击“确定”按钮回到数据流主设计面板。

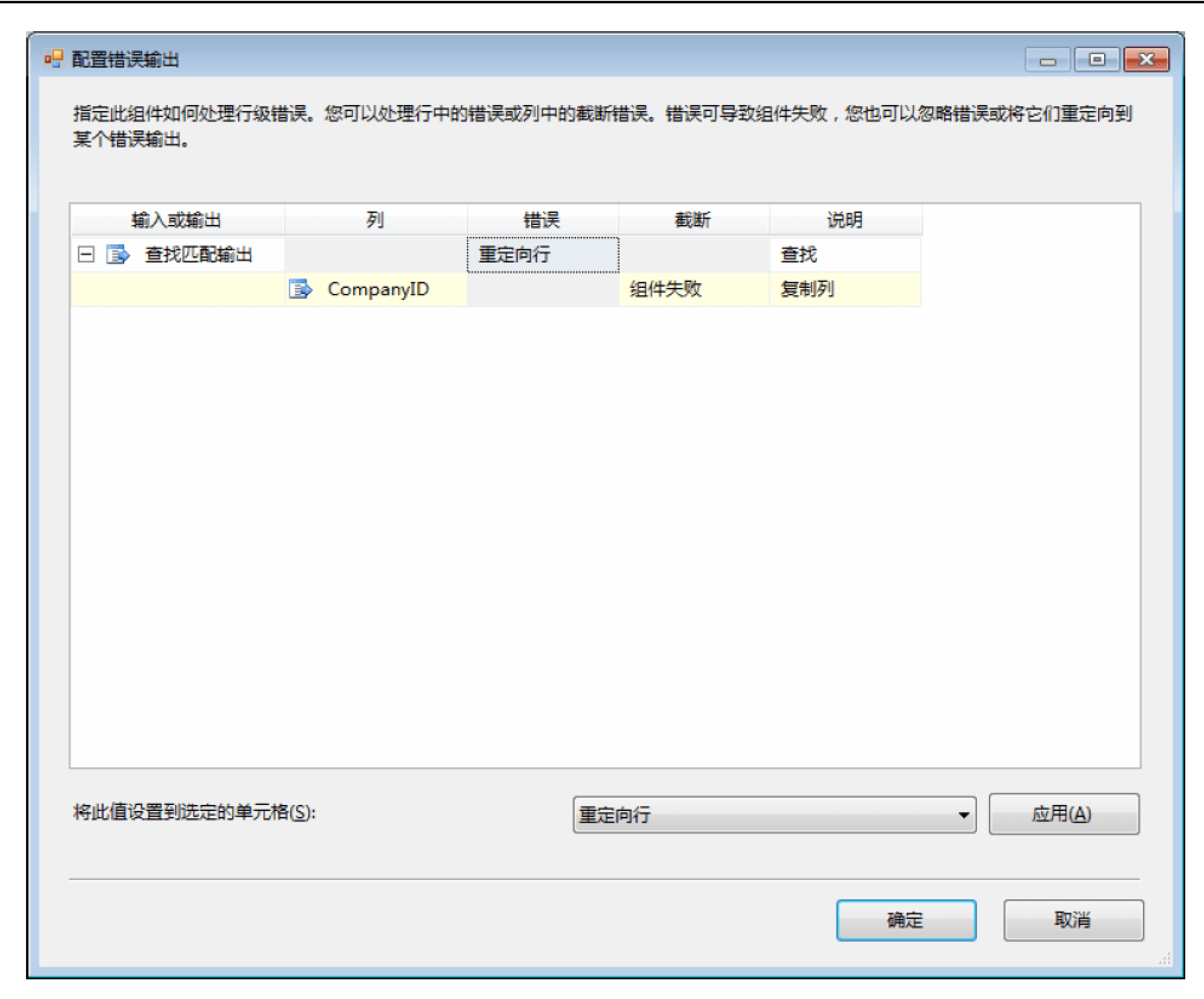


图 18.19　“配置错误输出”对话框

（4）双击“平面文件目标”组件，创建平面文件连接管理器，将所有输出内容保存到硬盘的某个文件中。

（5）运行修改后的 SSIS 包，可以看到整个流程全部成功完成，但是有 2 行数据在查找错误输出数据流中，如图 18.20 所示。

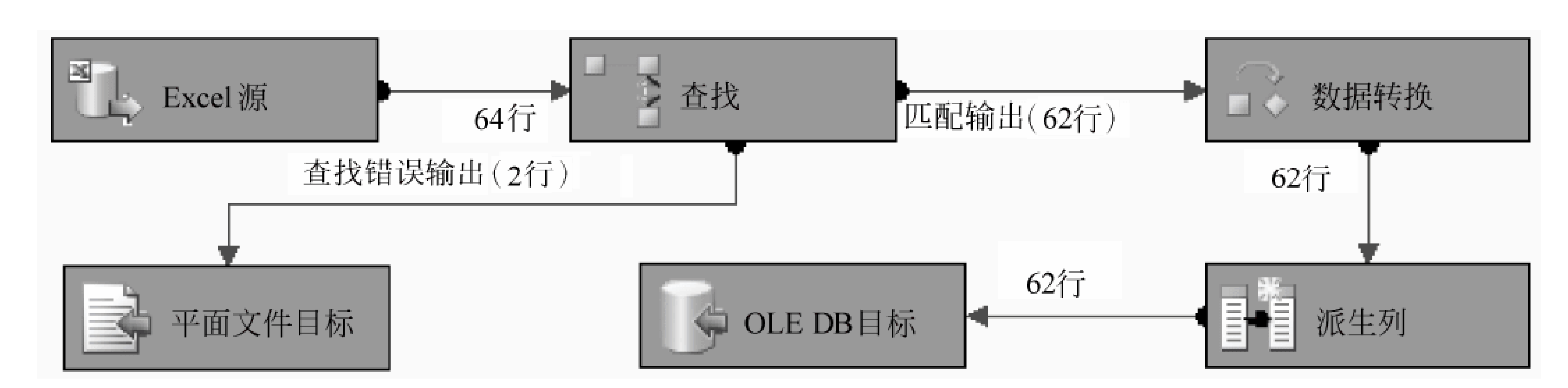


图 18.20　进行了错误处理的 SSIS 包

现在只需要再去查看文本文件中的内容，便可找到哪些数据不符合标准，然后再手动处理即可。

这里只是举了个简单的错误处理示例，在实际应用中可能会更复杂和自动，而不仅仅是将错误数据输出到文本文件中。

18.2.7　变量的使用

变量是 SSIS 体系结构中的一个亮点，通过变量允许在运行时动态控制程序包，就像

用户在.NET 语言中所使用的变量一样。变量分为两种类型，分别是系统变量和用户变量。

系统变量是建在 SSIS 中的变量，而用户变量则是由 SSIS 开发人员自己创建的。变量也可以作用于不同的域，默认作用域为整个程序包。作用域可以设置为容器、任务或者是程序包中的事件处理程序。在主设计面板中右击，在弹出的快捷菜单中选择“变量”选项，打开“变量”窗口，如图 18.21 所示，其中列出的都是系统变量。

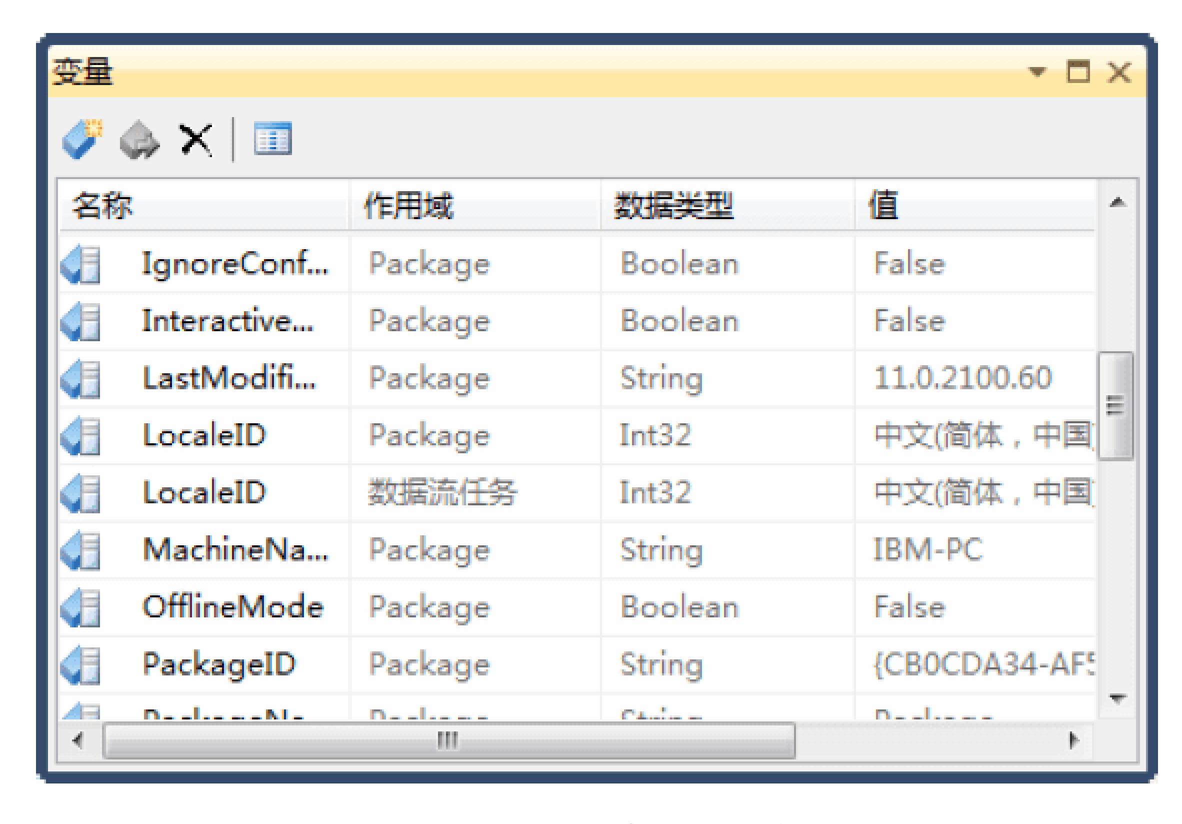


图 18.21　变量列表

StartTime、UserName 等系统变量常用于数据流任务中用于表示导入的时间、导入操作的用户等，同导入数据一起记录到目标中。

例如在 Budget 表中创建了 CreateTime 列用于记录数据导入的时间，则可以在派生列组件中将系统变量 StartTime 作为派生列添加到数据流中，如图 18.22 所示。

派生列名称	派生列	表达式	数据类型	长度	精度	小数位数	代码页
Status	<作为新列添加>	1	四字节带符号的整数 [D...				
Year	<作为新列添加>	YEAR([开始时间])	四字节带符号的整数 [D...				
CreateTime	<作为新列添加>	@[System::StartTime]	日期 [DT_DATE]				

图 18.22　使用系统变量作为派生列

系统变量是一个只读的变量，不能人为改变其值，如果需要读取和写入变量，则可以创建用户变量。

18.2.8　使用容器进行批量导入

容器是 SSIS 中为包提供结构及为任务提供服务的对象。它们支持包中的重复控制流，并且将任务和容器分组为有意义的工作单元。除了任务，容器还可以包含其他容器。

包将容器用于下列用途：

- 重复执行集合中每个元素的任务，这些元素包括文件夹中的文件、架构或 SQL 管理对象（SMO）等。例如，包可以运行驻留在多个文件的 Transact-SQL 语句中。
- 重复执行任务，直到指定表达式的求值结果为 false 为止。例如，包可以一周七次，每天一次发送一封不同的电子邮件。
- 将必须作为一个单元成功或失败的任务和容器分组到一起。例如，包可以将在数据库表中删除和添加行的任务分组到一起，然后当其中一个任务失败时提交或回滚所有任务。

Integration Services 提供了 4 种用于生成包的容器，下面列出了容器类型：

- Foreach 循环容器，通过使用枚举器重复运行控制流。
- For 循环容器，通过测试某个条件重复运行控制流。
- 序列容器，将任务和容器分组到那些作为包控制流子集的控制流中。
- 任务宿主容器，为单个任务提供服务。

现在业务人员从各个分公司收集了大量的物流数据，每一个分公司的数据都以一个 txt 文件存放，而且所有 txt 文件中内容的格式相同。现在需要把这些分公司物流数据导入到数据库中。显然，一个一个地修改文件连接来导入数据是费时费力的方法，这里可以使用 Foreach 容器实现批量数据的导入。下面是具体的实现方法。

（1）先按照一般的方法建立好一个数据流任务，保证这个数据流任务能够将一个物流 txt 文件导入到数据库中。

（2）在“工具箱”中，展开“控制流项”，然后将“Foreach 循环容器”选项拖曳到“控制流”选项卡的设计图面上。

（3）右击新添加的“Foreach 循环容器”图标，在弹出的快捷菜单中选择“编辑”选项，弹出 Foreach 循环编辑器对话框。在“常规”选项页上，在“名称”文本框输入 Foreach-FileinFolder 为该容器命名。

（4）在“集合”选项页上，选择“Foreach 文件枚举器”选项。在“枚举器配置”选项区域中，单击“浏览”按钮，弹出“浏览文件夹”对话框。在其中找到包含教程示例数据的示例数据文件夹。在“文件”文本框中，输入*.txt，如图 18.23 所示。

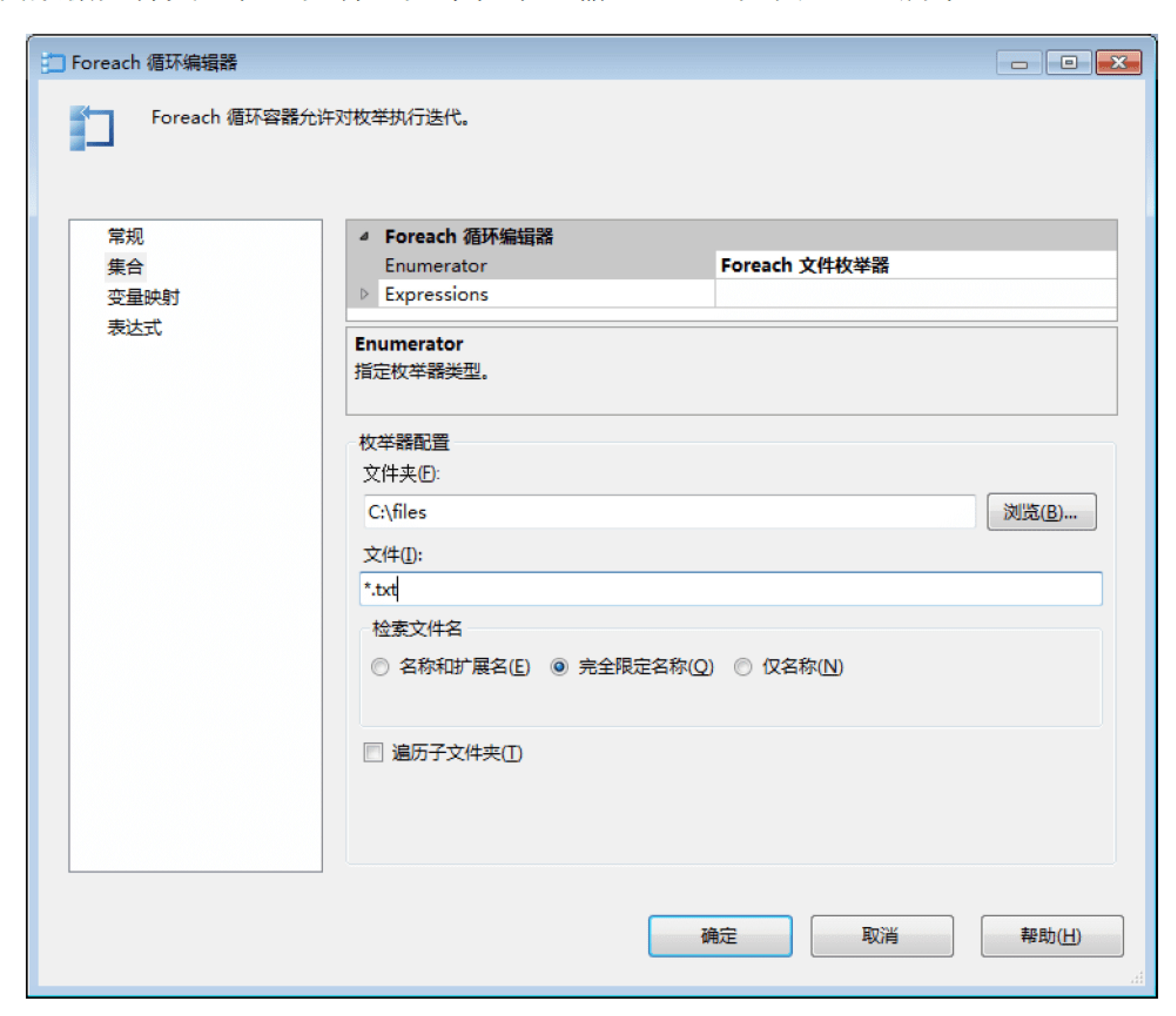


图 18.23　Foreach 循环编辑器集合设置

（5）选择“变量映射”选项卡切换到变量设置界面。在“变量映射”的“变量”列中，单击空单元格并选择“<新建变量…>”选项。在“添加变量”对话框中，在“名称”文本框输入 varFileName，类型为 string。

（6）单击“确定”按钮，退出“Foreach 循环编辑器”对话框。将前面做好的数据流任务拖曳到现已重命名为 ForeachFileinFolder 的 Foreach 循环容器中。

（7）修改平面文件连接管理器。在“连接管理器”窗格中，单击平面文件数据源。在“属性”窗口中，针对“表达式”，单击空单元，然后单击省略号按钮“(…)”。在“属性表达式编辑器”对话框的“属性”列中，输入或选择 ConnectionString 选项。在“表达式”列中，单击省略号按钮“(…)”以打开“表达式生成器”对话框。在“表达式生成器”对话框中，展开“变量”节点。将变量 User::varFileName 拖曳到“表达式”框中。单击“确定”按钮关闭“表达式生成器”对话框。再次单击“确定”按钮关闭“属性表达式编辑器”对话框。

（8）按下 F5 键运行该 SSIS 包，可以看到 Foreach 容器和数据流任务都变成了绿色，这说明已经将全部的数据都添加到数据库中了。

18.3 分析服务

在通过集成服务将各种异构平台的数据收集到 SQL Server 数据库中后，集成服务还可以对 SQL Server 中的数据进行进一步的处理，使之形成数据仓库。本节将主要讲解如何利用分析服务对数据仓库中的数据进行数据挖掘和联机分析处理。

18.3.1 分析服务简介

分析服务为商务智能应用程序提供联机分析处理（OLAP）和数据挖掘功能。在 OLAP 方面，分析服务允许设计、创建和管理包含从其他数据源（如关系数据库）聚合的数据多维结构，以实现对 OLAP 的支持。对于数据挖掘应用程序，Analysis Services 允许设计、创建和可视化处理那些通过使用各种行业标准数据挖掘算法，并根据其他数据源构造出来的数据挖掘模型。

对于 OLAP，分析服务允许开发人员在一个或多个物理数据源中定义一个称为统一维度模型（UDM）的数据模型，从而很好地组合了传统的基于 OLAP 分析和关系报表的各个最佳方面。基于 OLAP、报表以及自定义 BI 应用程序的所有最终用户查询，都将通过 UDM 访问基础数据源中的数据，UDM 便是 OLAP 的核心。

分析服务中提供了一组丰富的数据挖掘算法，业务用户可使用这组算法挖掘其数据以查找特定的模式和走向。这些数据挖掘算法可用于通过 UDM 或直接基于物理数据存储区对数据进行分析。

1. 数据挖掘

数据挖掘通常称为“从大型数据库提取有效、可信和可行信息的过程”。或者说，数据挖掘就是派生数据中存在的模式和趋势。这些模式和趋势可以被收集在一起并定义为挖

掘模型。生成挖掘模型是过程的一部分，此过程包括从定义模型要解决的基本问题到将模型部署到工作环境的所有事情。此过程可以使用下列 6 个基本步骤进行定义。

（1）定义问题。

（2）准备数据。

（3）浏览数据。

（4）生成模型。

（5）浏览和验证模型。

（6）部署和更新模型。

如图 18.24 所示为过程中每个步骤之间的关系，以及 SQL Server 2012 中可用于完成每个步骤的技术。

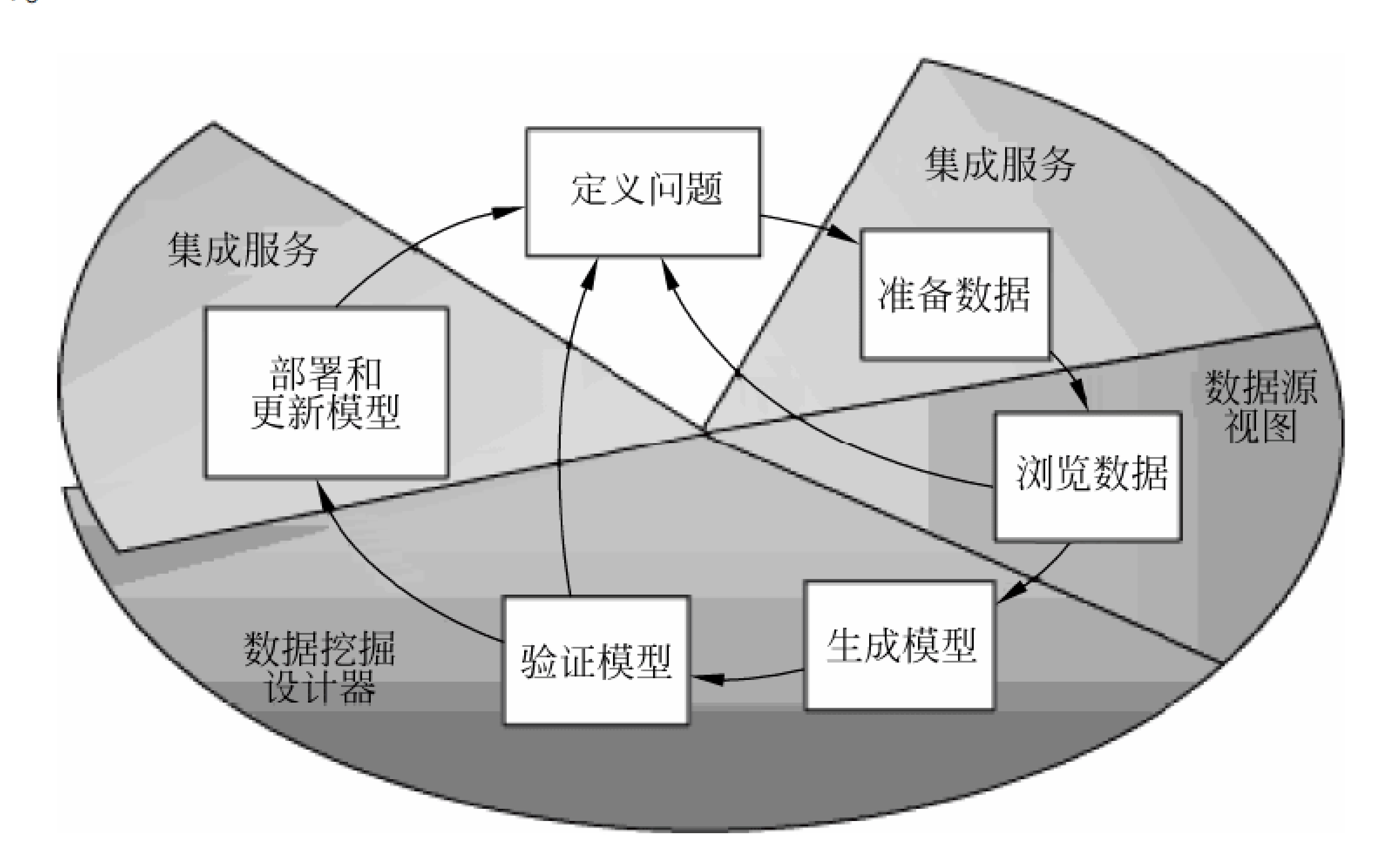


图 18.24　数据挖掘步骤

创建数据挖掘模型是一个动态、交互的过程。当创建了一个数据挖掘模型后，可能会发现数据不足，无法创建使用该挖掘模型，因此必须查找更多的数据。也可能会发现这些模型无法回答定义问题时所设定的问题，因此必须重新定义问题。所以关系图中所示的过程是一个循环过程，但是每个步骤并不需要直接执行到下一个步骤。因此，了解创建数据挖掘模型是一个过程，并且为了创建一个完美的模型，此过程中的每个步骤可能需要重复多次是非常重要的。

SQL Server 2012 提供用于创建和使用数据挖掘模型的集成环境，称为 Business Intelligence Development Studio。该环境包括数据挖掘算法和工具，使用这些算法和工具更易于生成用于各种项目的综合解决方案。

2. 多维数据

在进行多维数据分析时，所有 SSAS 维度都是基于数据源视图中的表列或视图列的属性组。独立于多维数据集存在的维度既可以在多个多维数据集中使用，也可以在一个多维数据集中多次使用，还可以在 Analysis Services 实例之间链接。独立于多维数据集存在的维度称为数据库维度，多维数据集中的数据库维度实例称为多维数据集维度。

简单维度对象由基本信息、属性和层次结构组成。其中的基本信息包括维度的名称、维度的类型、数据源和存储模式等。属性可定义维度中的实际数据。属性不一定属于层次

结构，但层次结构却要由属性生成。层次结构不但可创建级别的有序列表，还可定义用户浏览维度的方式。

简单 Cube 对象由基本信息、维度和度量值组组成。

- 基本信息包括多维数据集的名称、多维数据集的默认度量值、数据源和存储模式等。
- 维度是多维数据集中使用的实际维度组。所有维度都必须先在数据库的维度集合中定义，然后才能在多维数据集中引用。
- 度量值组是多维数据集中的度量值集。度量值组是具有常见数据源视图和维度集的度量值集合。度量值组是度量值的处理单元。可先对度量值组进行单独处理，然后再浏览。

注意：Microsoft SQL Server 2000 Analysis Services 中可用的专用维度在 Microsoft SQL Server Analysis Services 中不可用。

18.3.2 创建数据源和数据源视图

微软为了方便用户学习分析服务，创建了用于分析服务的示例数据库 Adventure Works DW2012，该数据库为通过集成服务从 Adventure Works 2012 数据库中提取的数据仓库。对于 Adventure Works DW2012 数据库，其中保存了大量产品信息、销售信息和客户信息，为了说明这一个个独立的表中存在着什么样的关系，下面就在 Analysis Services 项目中定义和部署这个多维数据集。假设需要分析的是客户区域和产品及时间各个维度上产品网上销量的关系，则创建数据源和数据源视图的操作如下。

（1）Analysis Services 项目的开发也是在 VS 2010 下进行的。打开 VS 2010，新建一个 Analysis Services 项目，并将项目命名为 SSAS1。这时 VS 2010 会为我们创建好 SSAS 文件夹，如图 18.25 所示。

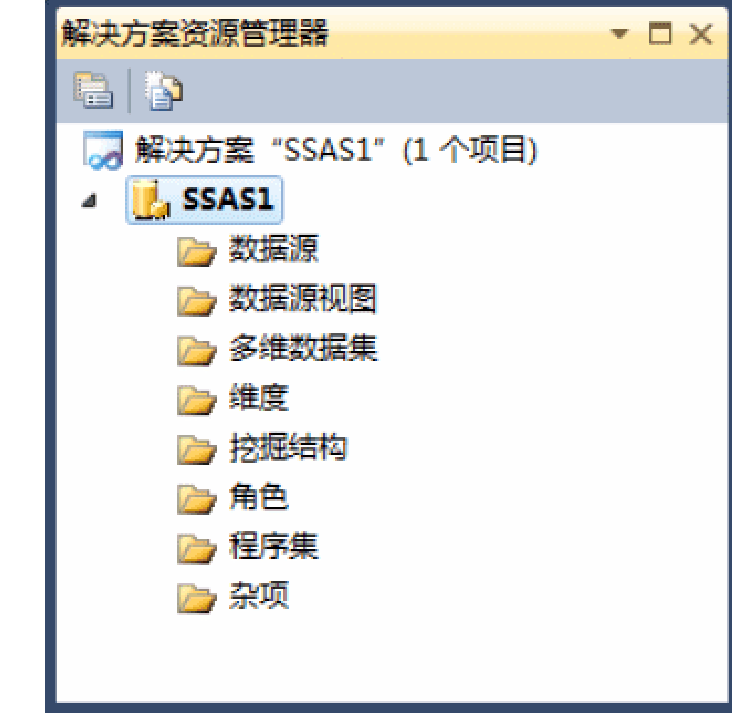


图 18.25 SSAS 的解决方案

（2）选择“数据源”文件夹，选择“新建数据源”选项，弹出“数据源视图向导”对话框。设置好要进行 SSAS 分析的数据库 Adventure Works DW2012，使用服务账户作为连接 SSAS 的凭据，单击“完成”按钮就将 Adventure Works DW2012 数据库添加到数据源中。

（3）选择“数据源视图”文件夹，在弹出的快捷菜单中选择“新建数据源视图”选项，弹出“数据源视图向导”对话框。使用上一步添加的数据源作为数据源视图的源。

（4）单击“下一步”按钮，进入选择表和视图界面。在“可用对象”列表中，可选择以下列表（同时按下 Ctrl 键可选择多个表）：

- DimCustomer 客户信息表；
- DimGeography 地理位置表；
- DimProduct 产品信息表；
- DimDate 时间码表；
- FactInternetSales 产品网上销售表。

然后单击“>”按钮，将选中的表添加到“包含的对象”列表中。如图 18.26 显示了将表添加到“包含的对象”列表后的“选择表和视图”页。

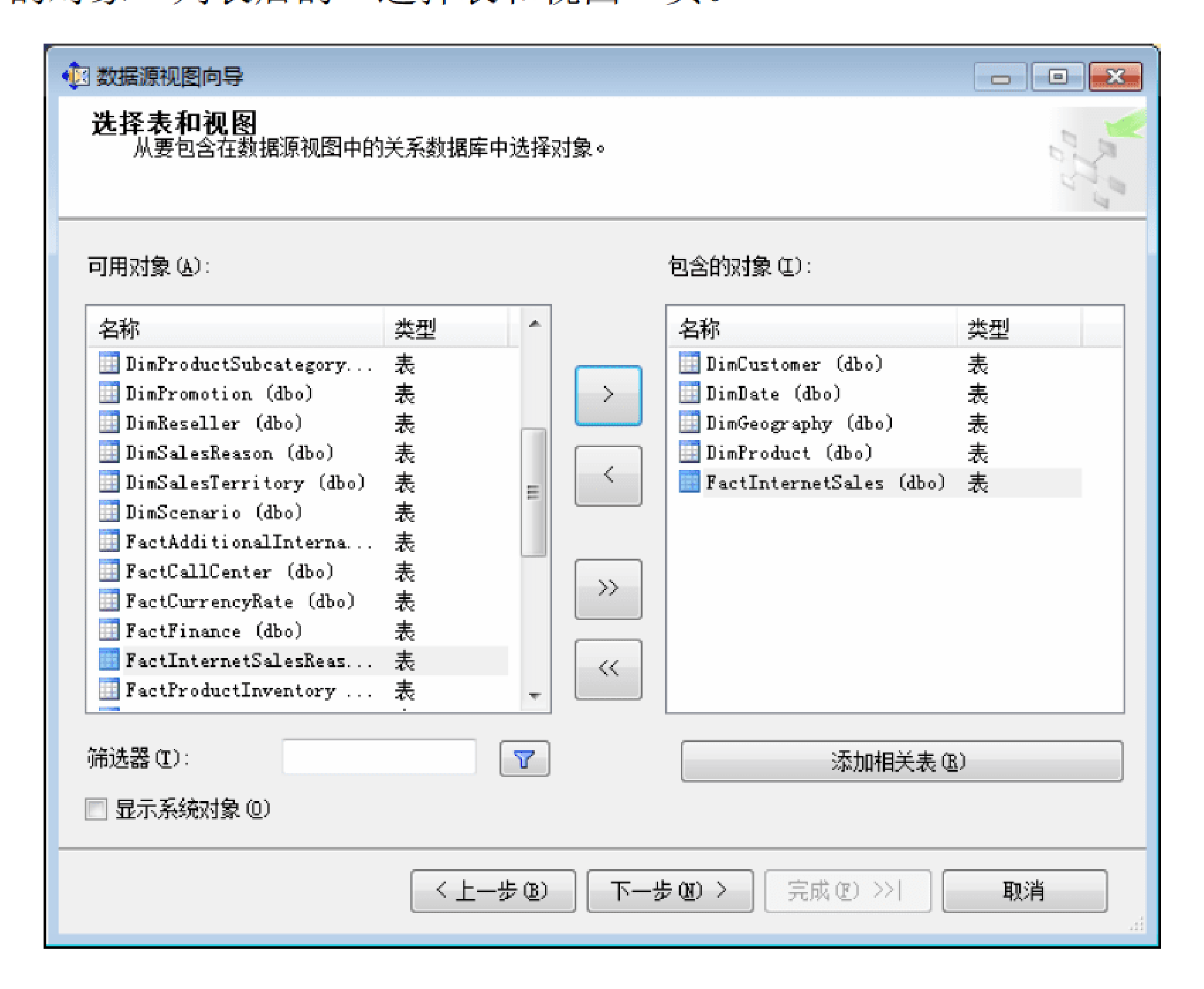


图 18.26　“数据源视图向导”对话框

（5）单击“下一步”按钮，然后单击“完成”按钮，系统将完成数据源视图的添加。完成后，VS 2010 会根据选择的表在数据库中的外键关系创建表的关系图，如图 18.27 所示。

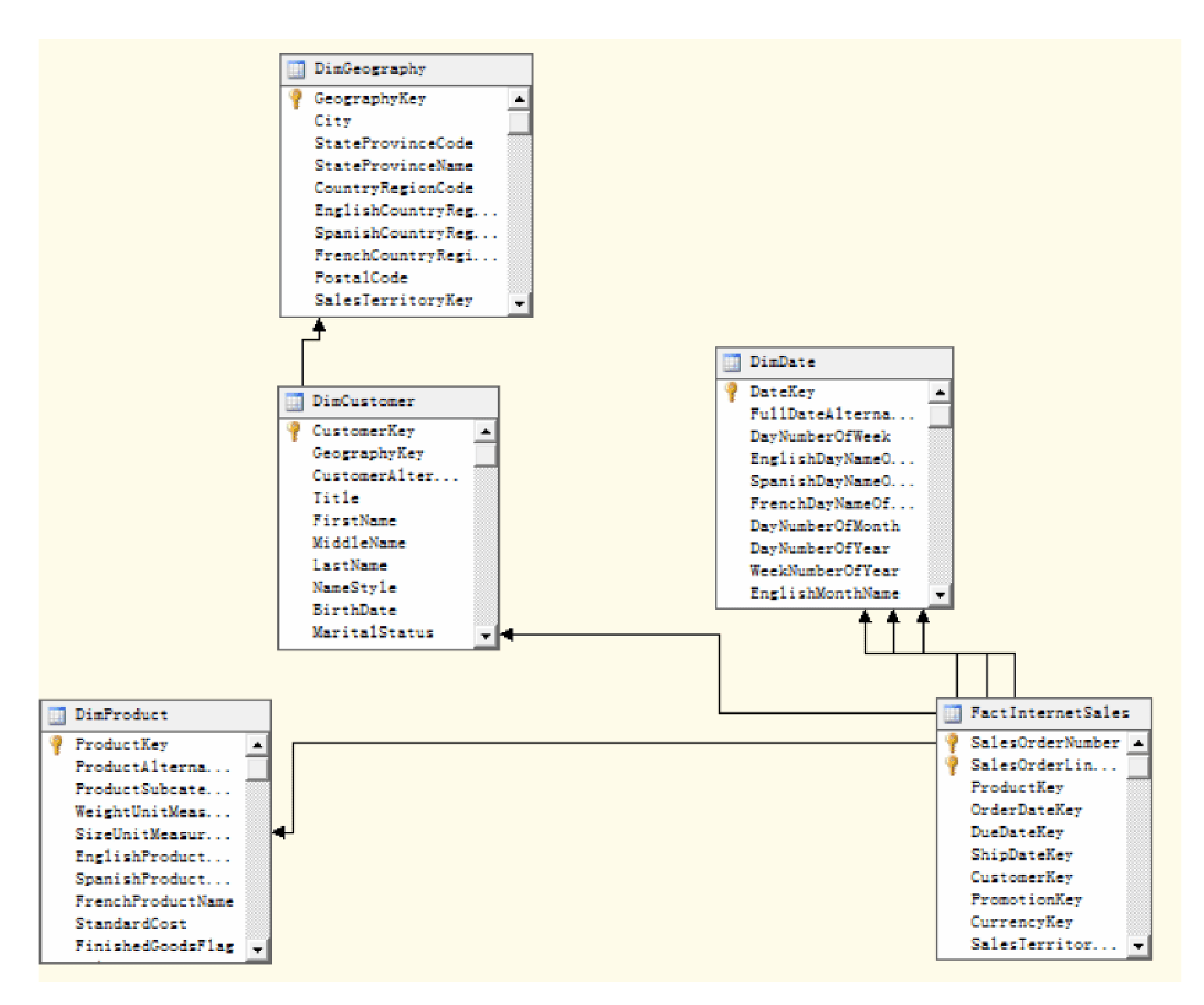


图 18.27　关系图

从关系图中可以看出，产品网上销售表包含有时间、客户、产品的外键，客户信息表包含有客户所在地理位置的外键。通过这些外键关系就可以建立它们直接的维度分析关系。

18.3.3　创建多维数据集

创建好数据源后接下来便可创建多维数据集，具体操作如下。

（1）选择解决方案资源管理器中的“多维数据集”节点，在弹出的快捷菜单中选择“新建多维数据集”选项，弹出“多维数据集向导”对话框。

（2）单击“下一步”按钮，进入创建方法选择界面，这里选择“使用现有表”选项，表示通过前面添加的数据源视图创建多维数据集。

（3）单击“下一步”按钮，进入度量值组表选择界面，单击“建议”按钮，这里使用 FactInternetSales 作为度量值组表。也就是说，无论从哪个维度进行分析，分析的核心数据都是该表产品的网上销售情况。

（4）单击“下一步”按钮，进入度量值选取界面。这里列出了 FactInternetSales 表下面的列，可选择需要进行分析的列。由于编号 Key 等字段在业务分析中并没有太大意义，所以可去掉与业务不相关的字段，如图 18.28 所示。

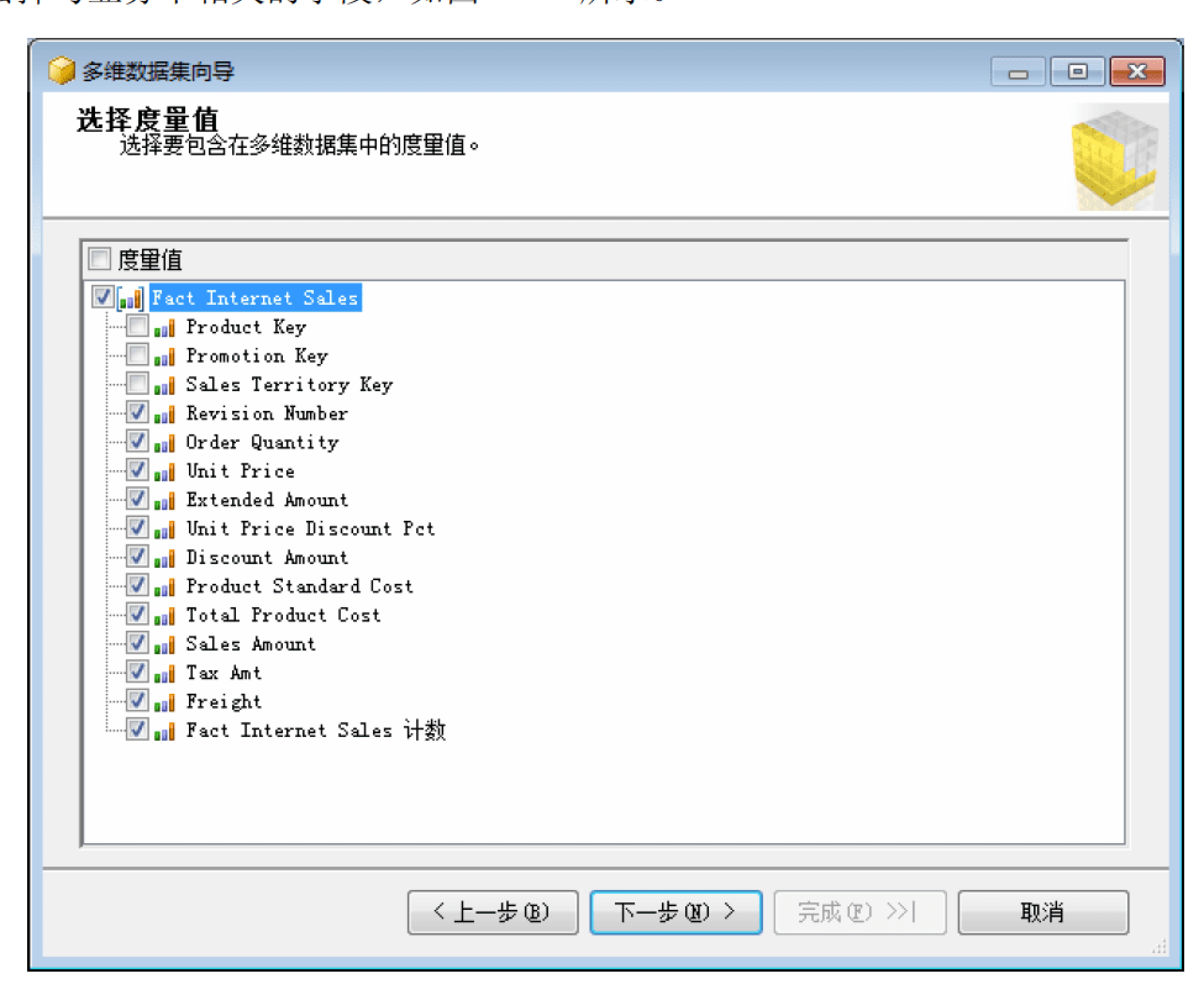


图 18.28　选择度量值

（5）再单击“下一步”按钮，即进入维度选择界面。这里列出的是要进行分析的维度，由于 FactInternetSales 表已经作为度量表，所以不需要再作为维度，取消对该表的选中，如图 18.29 所示。

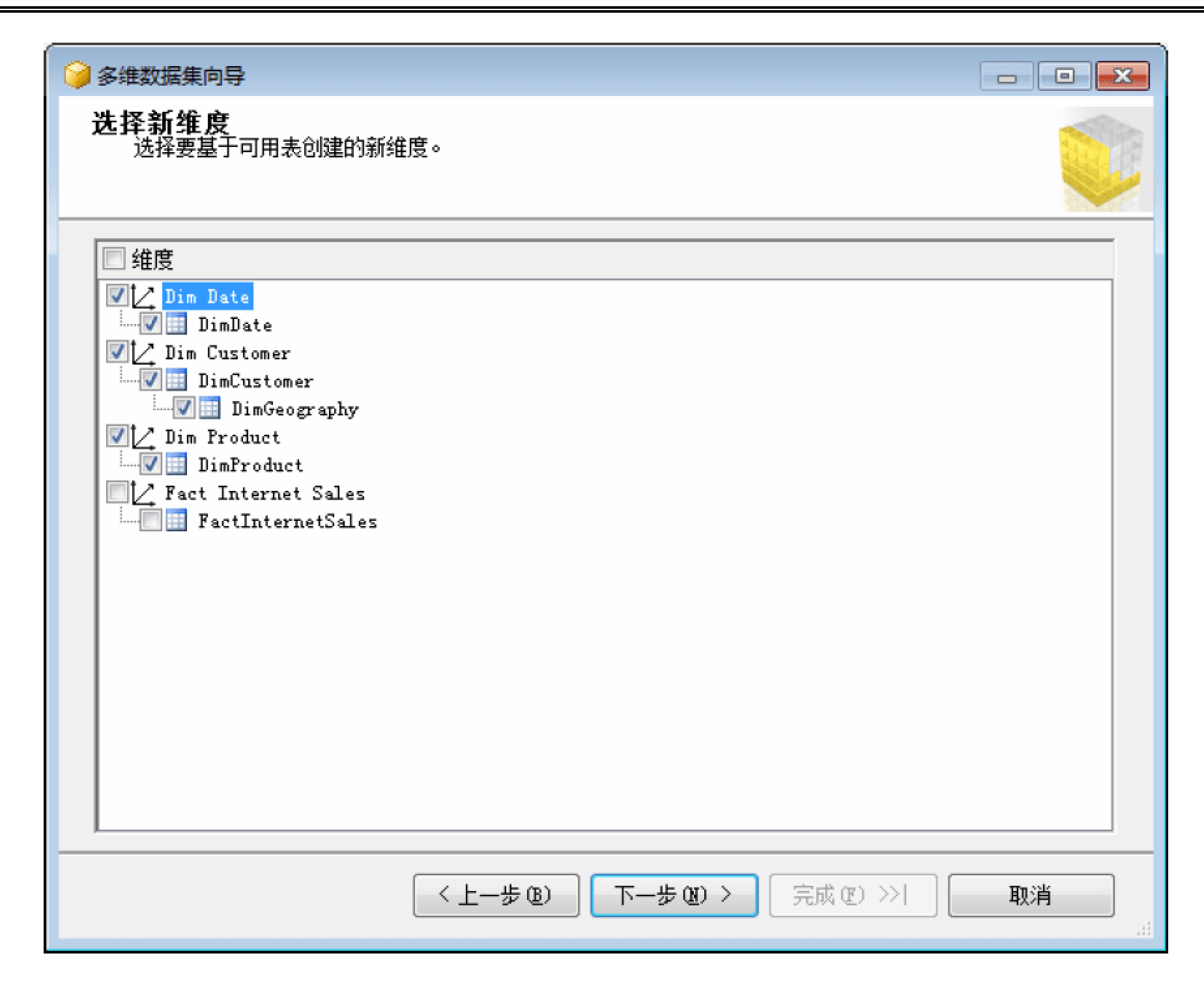


图 18.29　选择新维度

（6）单击“下一步”按钮，然后再单击“完成”按钮，系统将根据向导中的用户选择完成多维数据集的创建。在解决方案资源管理器的 SSAS1 项目中，创建的多维数据集显示在“多维数据集”文件夹中，而 3 个数据库维度则显示在“维度”文件夹中。此外，多维数据集设计器在开发环境的中央显示多维数据集关系。在此，在 VS 2010 的其他选项卡上已打开数据源视图设计器。如图 18.30 所示为该设计器中的维度表和度量值表。度量值表是用黄色表示，维度表用蓝色表示。

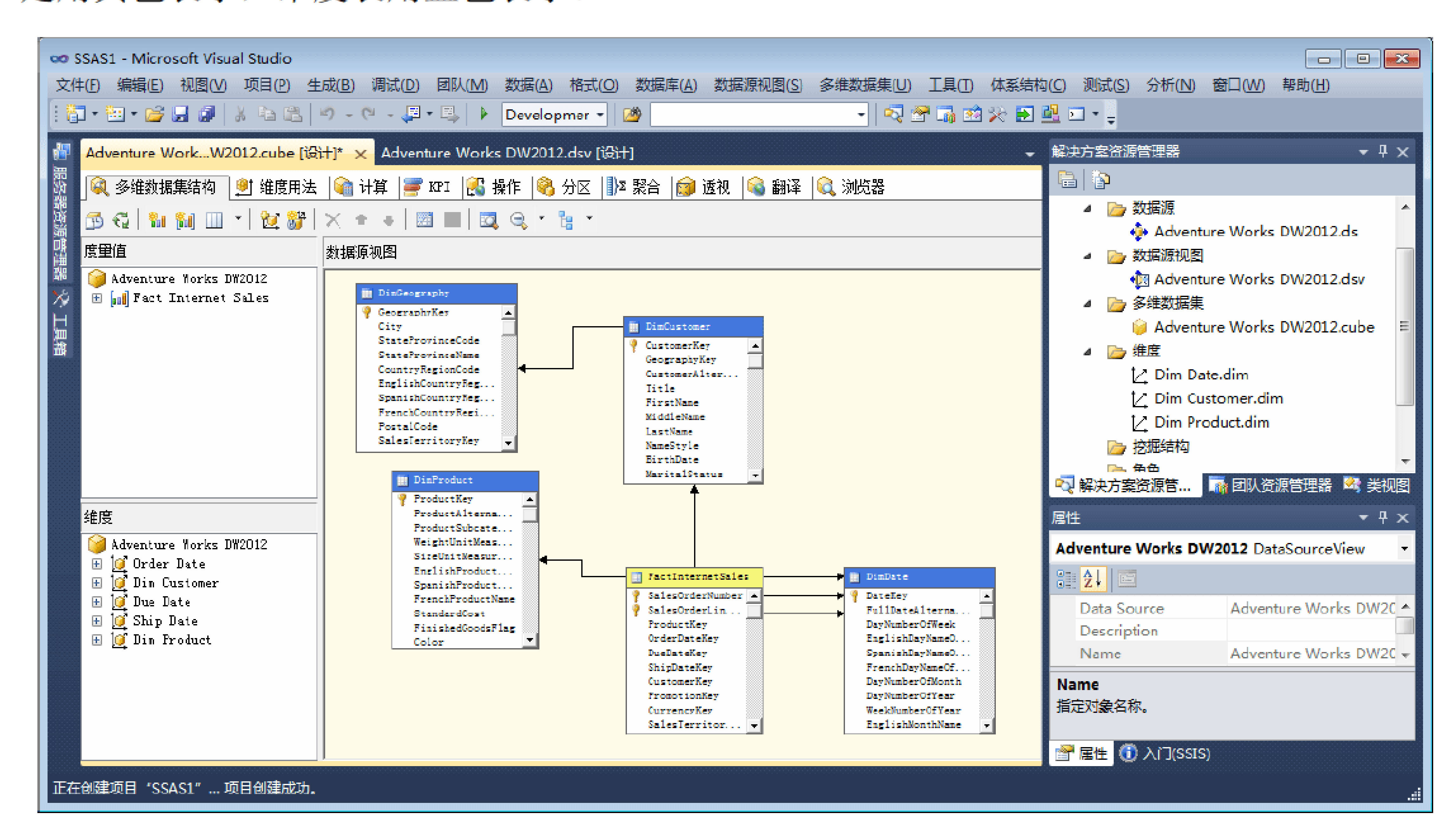


图 18.30　生成的多维数据集

（7）双击维度文件夹中的 Dim Product.dim 文件，打开 Product 维度设置界面。界面左侧为属性列表，右侧是数据源视图，若要设置维度的属性，只需要将需要的属性从数据源视图拖曳到左侧的属性列表中即可。这里 Large Photo 列是二进制列，所以不能添加，另外 Product Key 列作为该表的主键，默认是添加到属性中，所以也不需要重复添加，其他列则全部添加到属性中，添加后如图 18.31 所示。

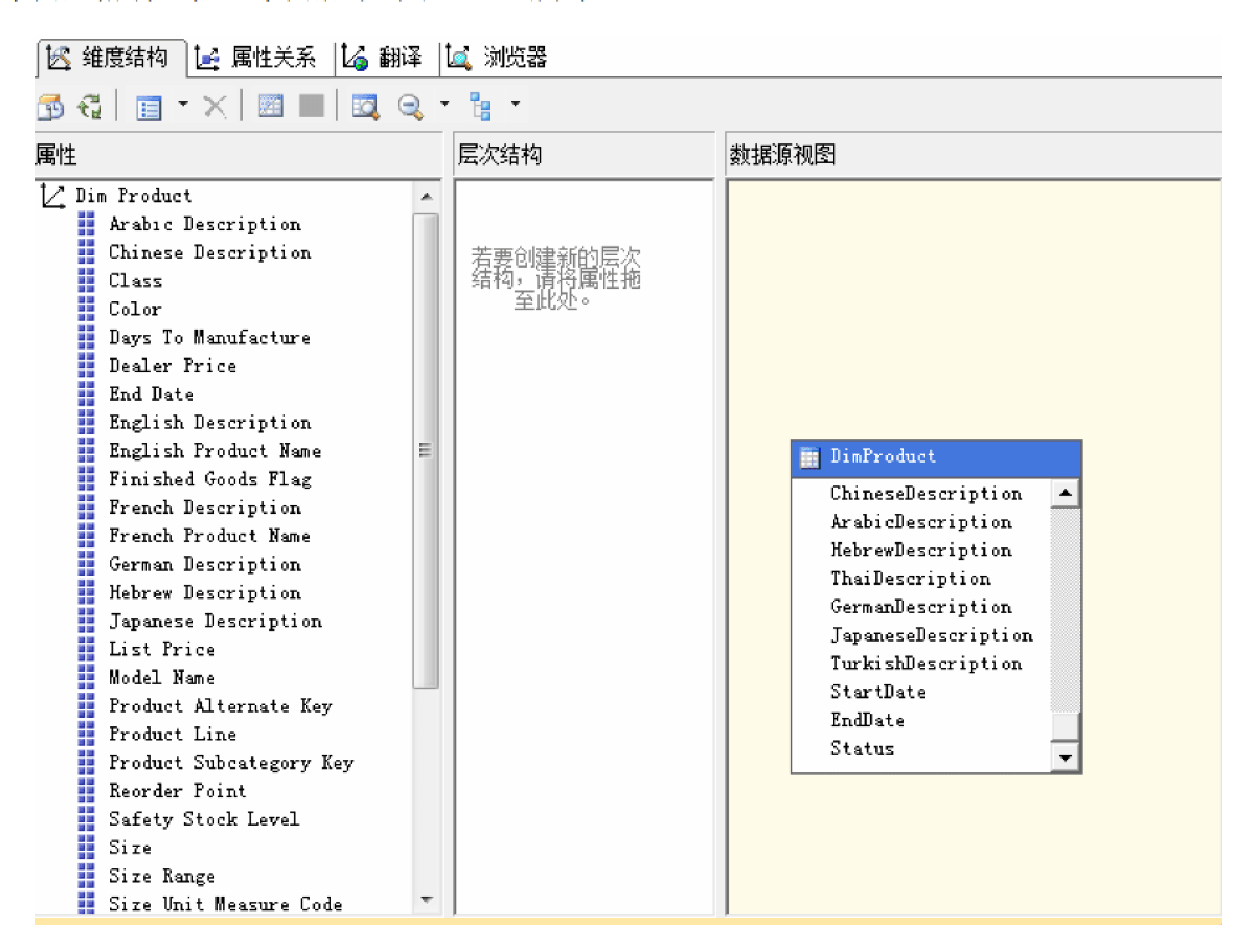


图 18.31　设置 Product 维度

（8）打开 Dim Date.dim 文件，该维度应该是一个时间维度。在 SSAS 中，时间维度是指其属性表示时间段（如年、半年、季度、月和天）的维度类型。时间维度中的时间段可提供用于分析和报告的基于时间的粒度级别。属性将按层次结构进行组织，并且时间维度的粒度主要由历史数据的业务和报表需求而决定。在属性选项卡中修改 Dim Date 的 Type 属性为“Time”标识出该维度是时间维度。

（9）接下来设置 Dim Date 的属性，这里用到了年、半年、季度、月、日期，将这几个属性对应的列 CalendarYear、CalendarSemester、CalendarQuarter、EnglishMonthName 和 FullDateAlternateKey 从数据源视图拖曳到属性中。

（10）由于这几个属性之间存在层次关系，所以需要将这几个属性按照顺序拖曳到层次结构面板中，最终 Dim Date 的设置如图 18.32 所示。

（11）打开 Dim Customer.dim 文件，由于 Customer 表与 Geography 表关联，所以在数据源视图中有 2 个表。将数据源视图的所有列拖曳到属性中，但是属性不要重复。在层次结构中将 State Province Name、Geography Key 和 Customer Key 依次拖入，形成层次关系。

（12）切换到“属性关系”选项卡，将关系修改为 Customer Key 关联 Geography Key，然后 Geography Key 关联 State Province Name，如图 18.33 所示。

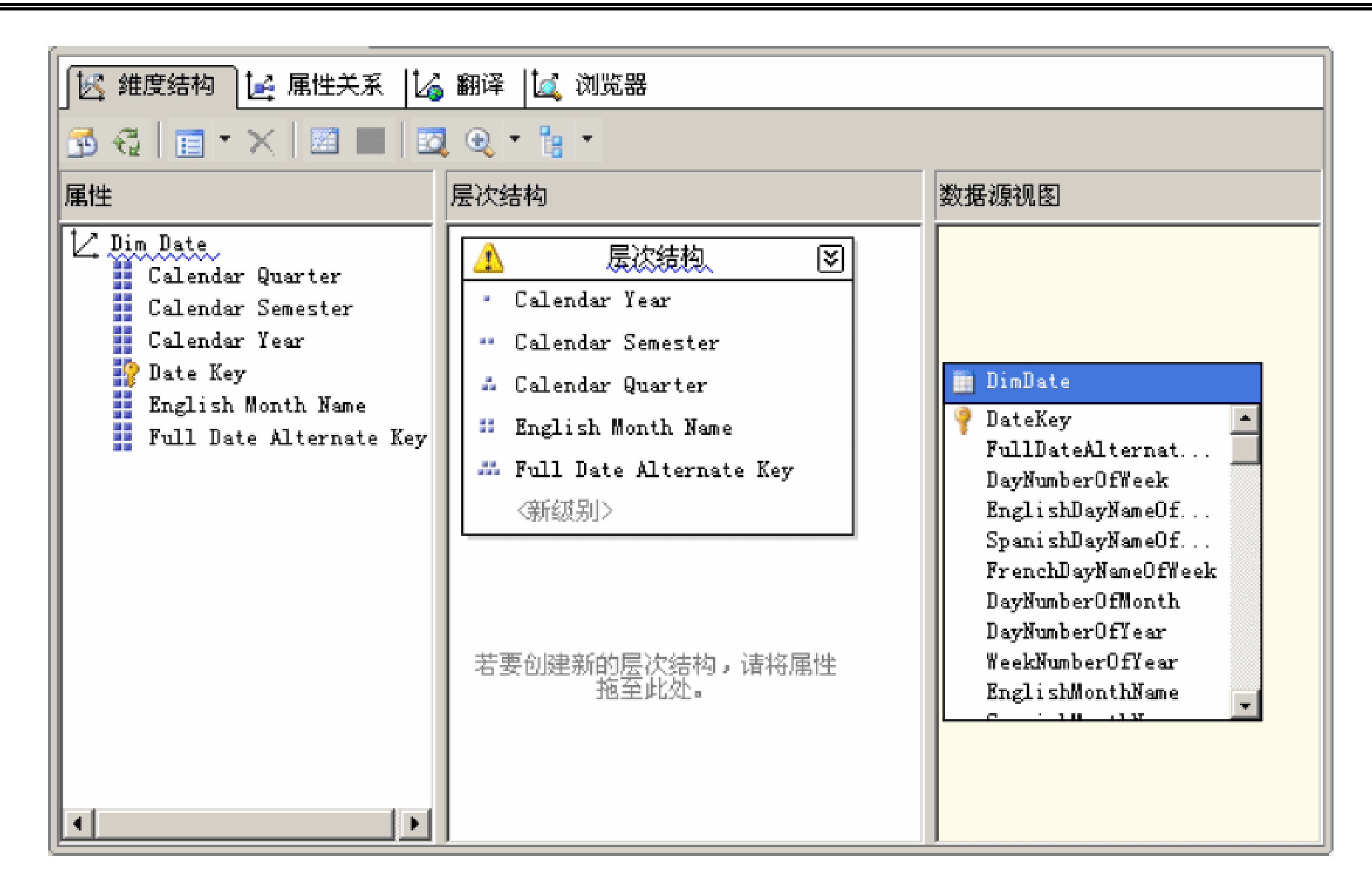


图 18.32　Dim Date 维度设置

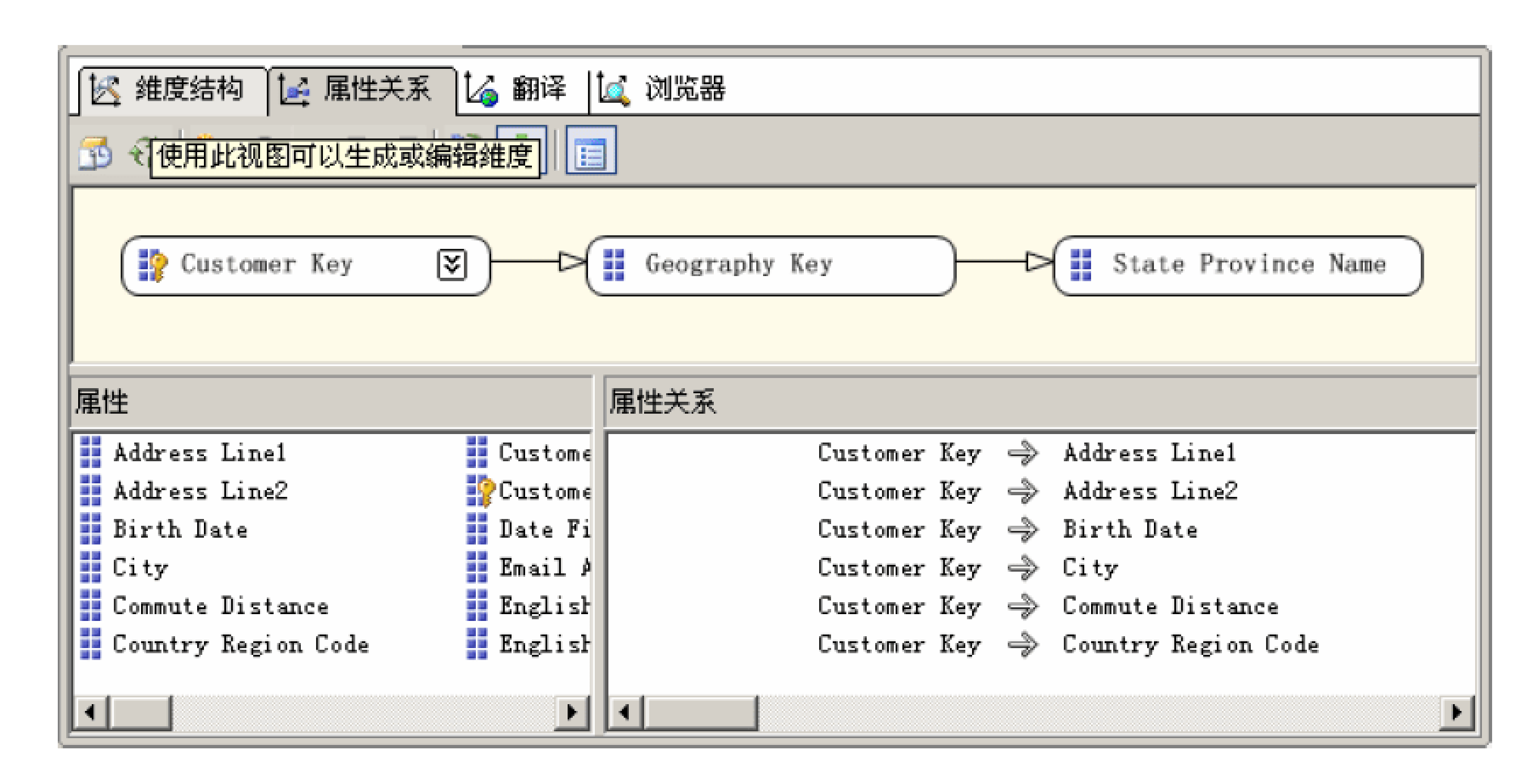


图 18.33　属性关系

（13）将所有对维度的修改保存到硬盘上，至此多维数据集已经成功创建完成。

18.3.4　部署分析服务

经过前面的操作，多维数据集已经创建完成，接下来就需要将其部署到服务器上，以进行维度分析。首先要确保用于部署的 SQL Server 2012 服务器的 Analysis Services 服务运行正常，如果没有运行，可以打开 SQL Server 配置管理器，启动对应的服务。

接下来回到 VS 2010 界面，指定 SSAS 项目需要发布到哪个服务器。右击 SSAS1 项目，在弹出的快捷菜单中选择“属性”选项，弹出“SSAS1 属性页”对话框，显示活动（开发）配置的属性，如图 18.34 所示。

可以定义多个配置，每个配置可以具有不同的属性。例如，不同的开发人员可能需要将同一项目配置为部署到不同的计算机上，并具有不同的部署属性，如不同的数据库名称或处理属性。

在图 18.34 的左窗格“配置属性”节点中，选择“部署”选项，查看项目的部署属性。

默认情况下，Analysis Services 项目模板将 Analysis Services 项目配置为将所有项目增量部署到本地计算机上的默认 Analysis Services 实例，以创建一个与此项目同名的 Analysis Services 数据库，并在部署后使用默认处理选项处理这些对象。

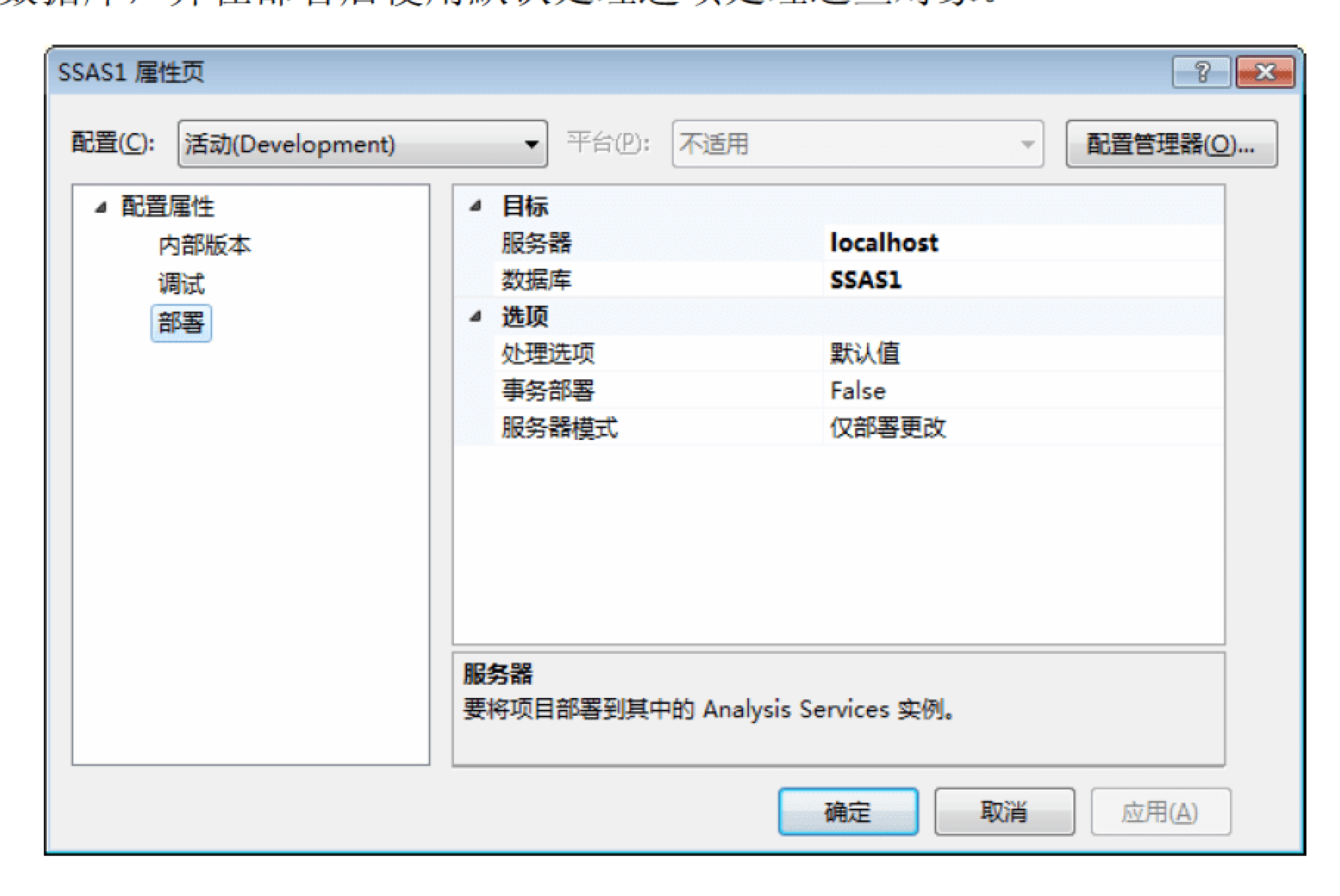


图 18.34　SSAS 项目属性

注意：如果要将项目部署到本地计算机上的命名 Analysis Services 实例或远程服务器上的实例，需要将“服务器”属性更改为相应的实例名，如<服务器名>\<实例名>。这里若部署的是远程服务器，本地机器必须与远程服务器处在同一个域中，而且当前用户有登录 Analysis Services 实例的权限。Analysis Services 服务只支持 Windows 认证方式，不支持用户名密码认证。

确定了 Analysis Services 服务器后，单击“确定”按钮，再回到 VS 2010 中。在解决方案中右击 SSAS1 项目，在弹出的快捷菜单中选择“部署”选项，系统将自动保存当前项目，然后将该项目部署到设置好的 Analysis Services 服务器上。同时弹出“部署进度”对话框，显示当前运行状态。部署完成后如图 18.35 所示。

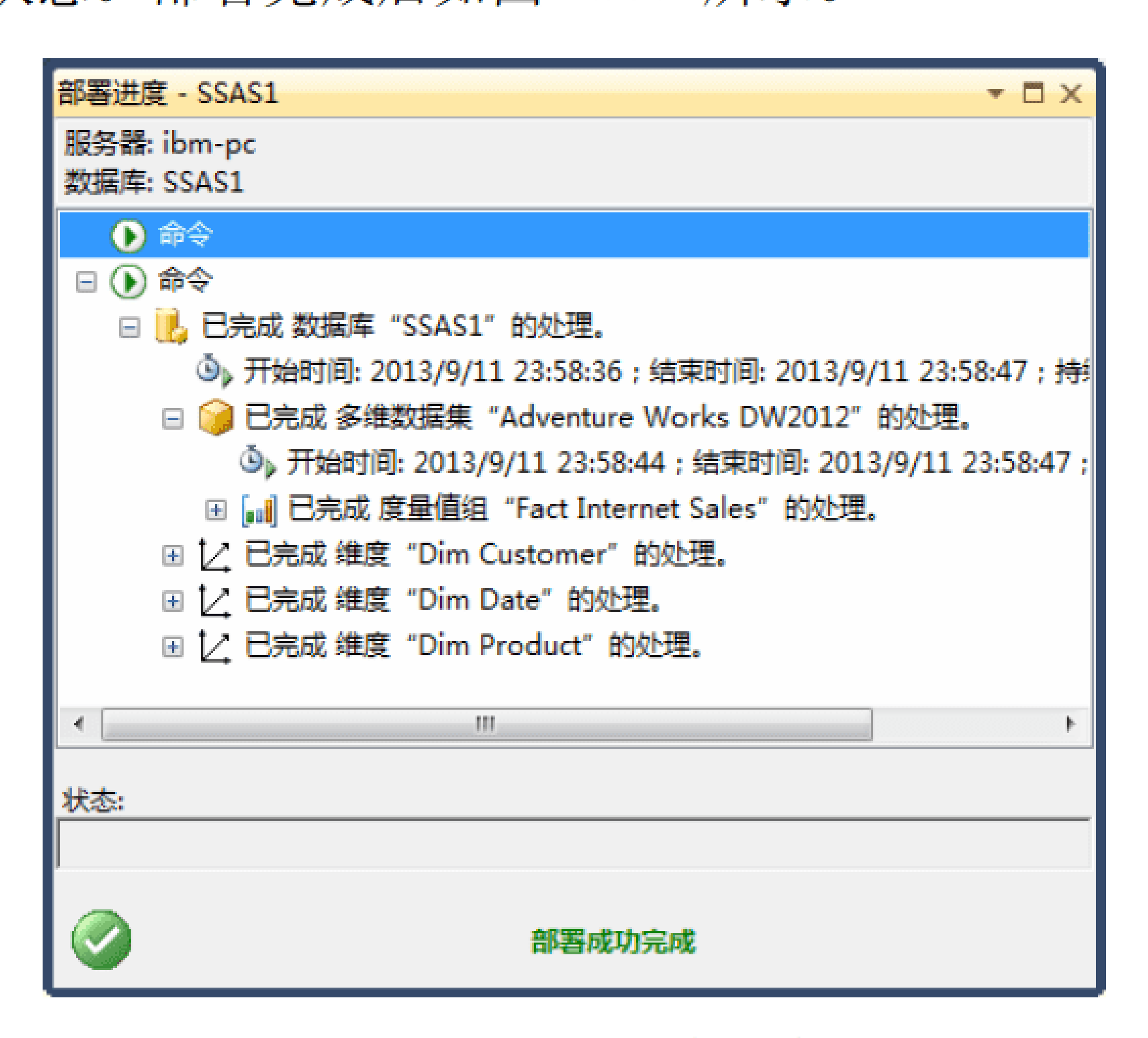


图 18.35　部署分析服务

这时如果使用 SSMS 登录 Analysis Services 服务器，登录后的对象资源管理器如图 18.36 所示，可以看到 SSAS1 实例已经在服务器上了。

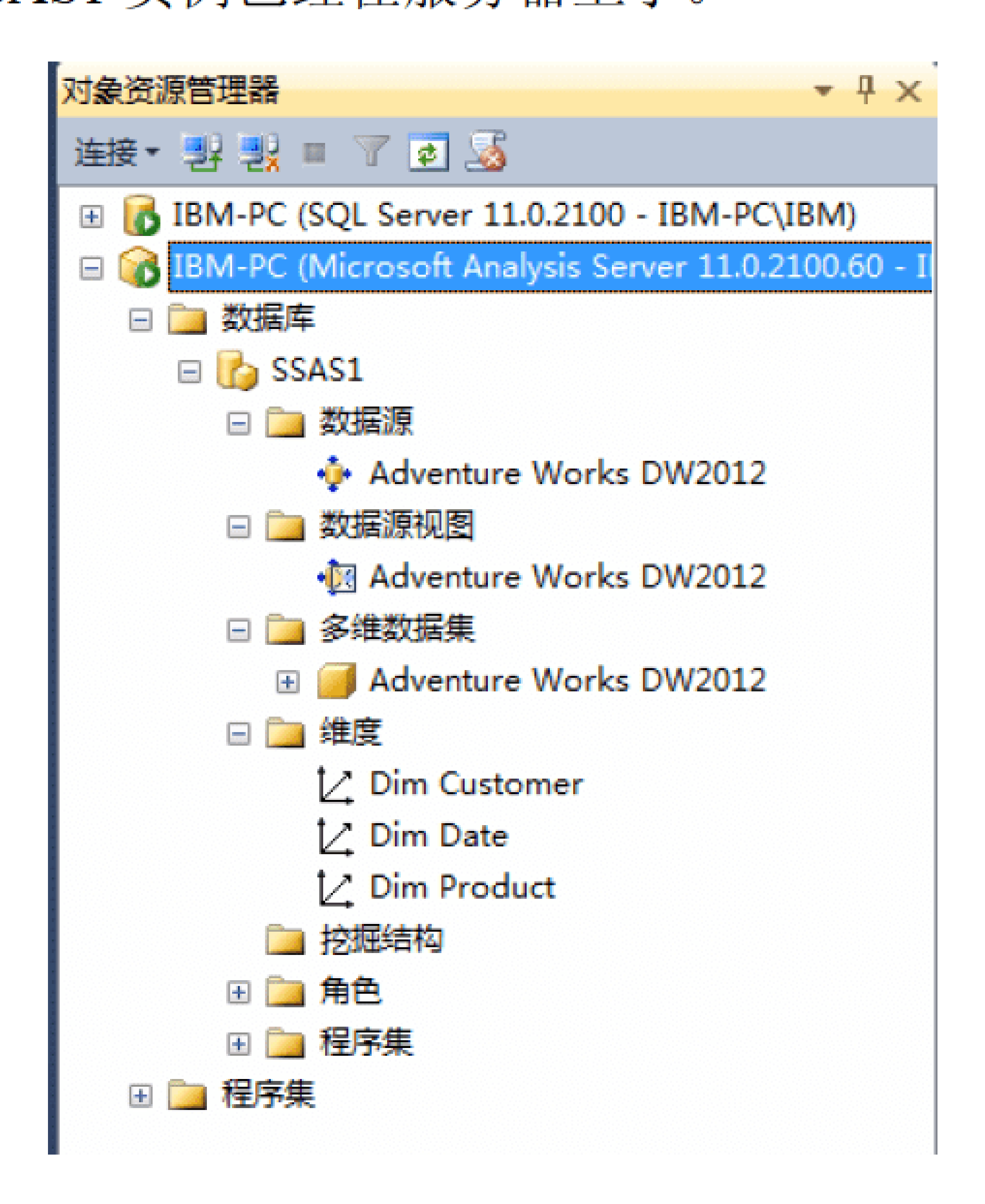


图 18.36　SSMS 连接 Analysis Services 服务器

18.3.5　显示分析数据

在将多维数据集成功部署到 Analysis Services 的本地实例后，便可以浏览多维数据集中的实际数据。

在 Visual Studio 2010 中打开 Dim Date.dim 文件，切换到浏览器选项卡，可以通过层次结构的树状形式查看 Date 维度的信息。在层次结构下拉列表框中选择“层次结构”选项，系统将根据在维度设计时层次结构的定义来显示 Date 维度的数据。

展开 All 级别成员以显示 CalendarYear 级别的成员，展开 2005 成员以显示 CalendarSemester 级别的成员，展开 2 成员以显示 CalendarQuarter 级别的成员，展开 3 成员以显示 English Month Name 级别的成员，展开 August 成员以显示 FullDateAlternateKey 级别的成员。展开后的结果如图 18.37 所示。

双击打开多维数据集 Adventure Works DW2012.cube，切换到“浏览器”选项卡，在该设计器的左窗格将显示多维数据集的元数据。“透视”和“语言”选项显示在“浏览器”选项卡的工具栏上。另外还会看到“浏览器”选项卡在右侧包含两个“元数据”窗格：上面的窗格是“筛选器”窗格，下面的窗格是“数据”窗格。

例如要分析每个城市的产品销售数量，则在元数据窗格中，依次展开度量值、Internet 销售，然后将销售数量值 Sales Amount 拖到数据窗格的区域。之后在元数据窗格中，展开 Dim Customer 维度，将 City 拖曳到数据窗格区域，系统将列出所有城市的产品销售数量，如图 18.38 所示。

这里只是从城市这一个维度来分析，如果要知道每个城市对产品颜色的喜好情况，即每个城市的每种颜色产品的销售数量，则需要添加颜色维度。从元数据窗格中展开 Dim

Product 维度，将其下的 Color 拖曳到数据窗格区域，系统将以城市和颜色两个维度来分析产品销量，如图 18.39 所示。

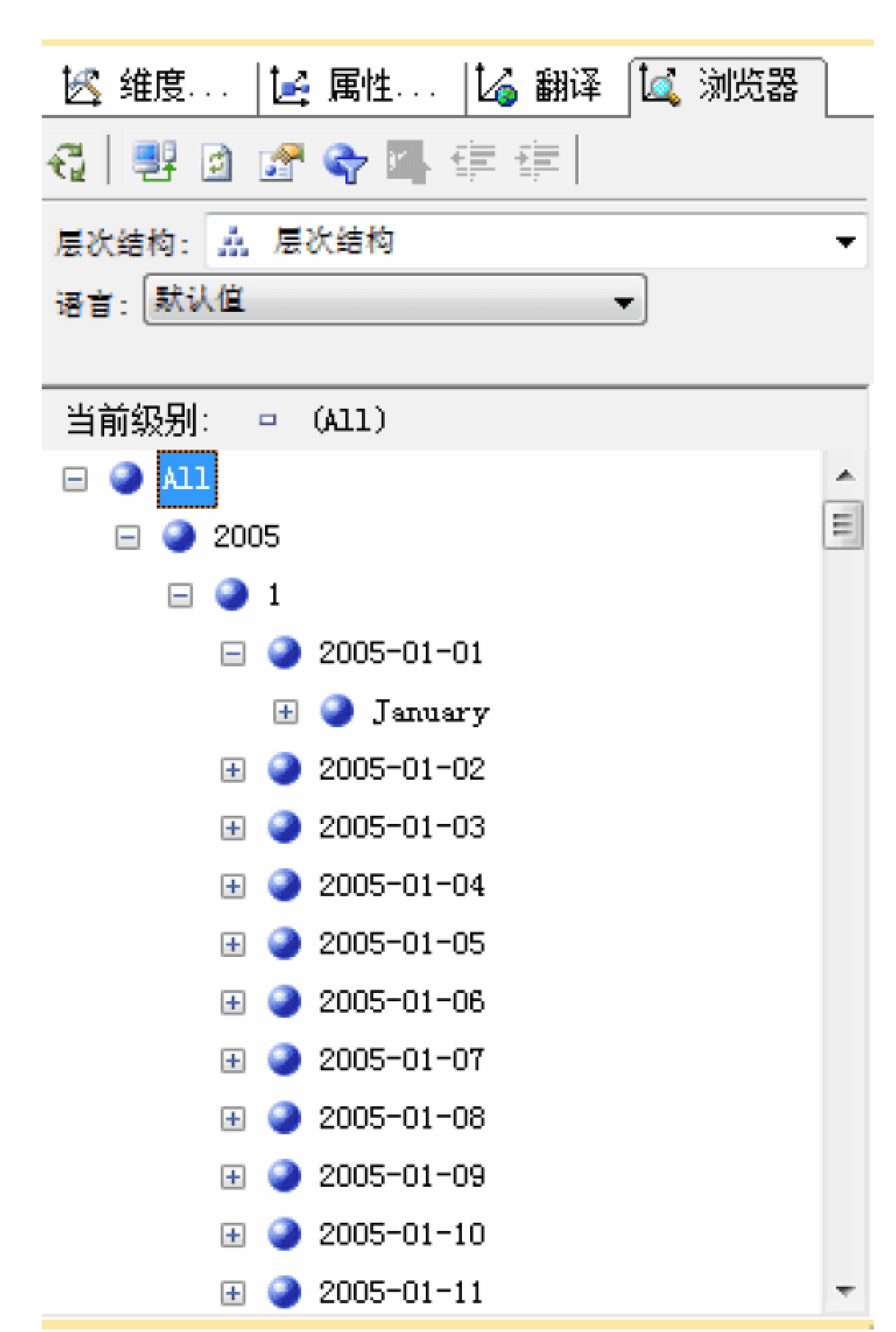


图 18.37　维度浏览器

City	Sales Amount
Ballard	45308.9957
Barstow	3578.27
Basingstoke H...	3271.6782
Baytown	25.48
Beaverton	161959.427
Bell Gardens	5920.24
Bellevue	2049.0982
Bellflower	302278.805699997
Bellingham	207613.253199999

图 18.38　分析城市与销量

City	Color	Sales Amount
Ballard	Black	16184.7282
Ballard	Blue	5308.54
Ballard	Multi	335.9
Ballard	NA	1318.26
Ballard	Red	14276.06
Ballard	Silver	879.47
Ballard	White	8.99
Ballard	Yellow	6997.0475
Barstow	Red	3578.27

图 18.39　分析城市、颜色与销量

注意：对于每一个维度，系统都将有一个汇总结果显示在表中，折叠到上一个层次结构后显示的就是汇总的结果。

通过拖曳不同的字段到数据窗格中将会产生不同维度的分析结果，了解了如何从不同维度获得分析结果后，接下来就是要将分析结果呈现出来，这就要用到报表服务。

18.4　报 表 服 务

用户可使用报表进行信息沟通、制定决策和识别机遇。报表服务（SQL Server 2008 Reporting Services，SSRS）是基于服务器的报表平台，它提供各种现成可用的工具和服务，帮助用户方便、快捷地创建、部署、管理和使用报表。

18.4.1　报表服务简介

报表服务是一种基于服务器的解决方案，用于生成从多种关系数据源和多维数据源提取内容的企业报表。报表服务一般集成在 IIS 中，通过 Web 方式进行管理，通过 Web 方式进行管理，并且通过 Web 方式可以发布以各种格式查看的报表。SQL Server 提供的报表可以通过基于 Web 的连接进行查看，也可以作为 Microsoft Windows 应用程序的一部分或 SharePoint 门户进行查看。报表服务包括下列核心组件：

- 一整套工具，可以用来创建、管理和查看报表。
- 一个报表服务器组件，用于承载和处理各种格式的报表。输出格式包括 HTML、PDF、TIFF、Excel、CSV 等。
- 一套 API，使开发人员可以在自定义应用程序中集成或扩展数据和报表处理，或者创建自定义工具生成和管理报表。

报表服务在生成报表时所使用的数据，可以基于 SQL Server、Analysis Services、Oracle 或任何 Microsoft.NET 数据访问接口（如 ODBC 或 OLEDB）提供的关系数据或多维数据。使用报表服务可以创建表格、矩阵和自由格式的报表。还可以创建使用预定义模型和数据源的即席报表。

在报表服务中生成的报表包括交互功能和基于 Web 的功能，在外观和功能上超越了传统的报表。虽然报表服务已与 Microsoft 的其他技术进行了集成，提供了强大的数据和图表展示，但是开发人员和第三方供应商可以开发其他相应的组件，以支持其他报表输出格式、传递格式、身份验证模式和数据源类型。在模块设计中特意创建了开发和运行时体系架构，以支持可能采用的第三方扩展和集成。

18.4.2　报表设计

和其他的 BI 设计一样，Reporting Services 的设计也是在 VS 2010 中完成的。在前面的 SSAS 分析中，已经得到了很多重要的数据，在这个环节，就使用 SSRS 将得到的数据以报表的形式展现出来。例如要创建一个报表显示每个城市的每种颜色产品的销售数量，则对应的操作如下。

（1）打开 VS 2010，新建报表服务器项目，项目名为 SSRS1。系统将在解决方案资源管理器中创建共享数据源文件夹和报表文件夹。

（2）选择“共享数据源”文件夹，在弹出的快捷菜单中选择“添加新数据源”选项，弹出“共享数据源属性”对话框，在“名称”文本框输入数据源名称 SSAS1，类型为 Microsoft SQL Server Analysis Services，然后单击“编辑”按钮设置连接字符串，设置好的数据源属

性如图 18.40 所示。

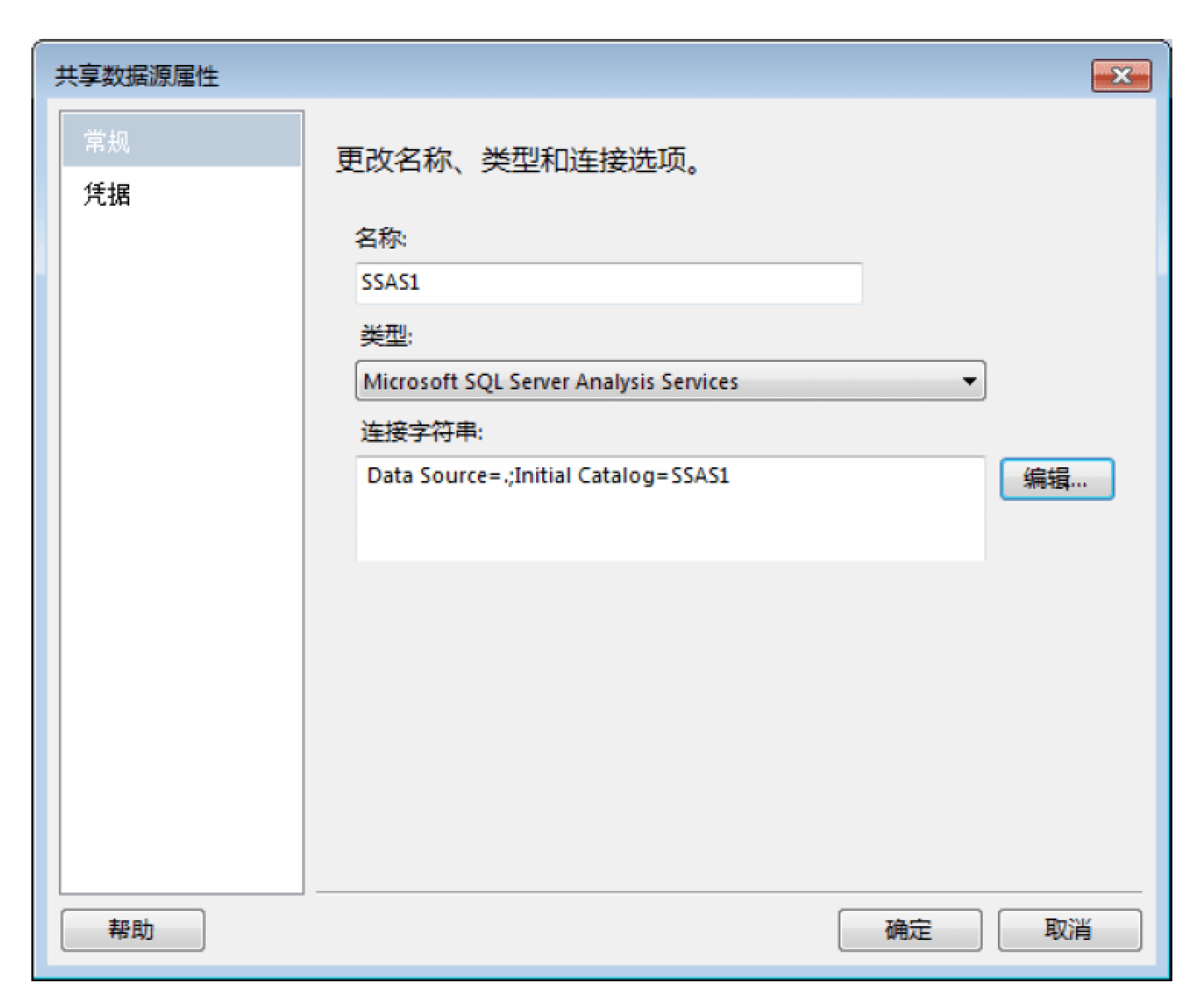


图 18.40 “共享数据源属性”对话框

（3）选择“报表”文件夹，在弹出的快捷菜单中选择“添加新报表”选项，弹出报表向导。单击“下一步”按钮，然后选择共享数据源“SSAS1”作为报表的数据源。

（4）单击“下一步”按钮进入设计查询界面，单击其中的“查询生成器”按钮，打开“查询设计器”对话框。

（5）将要查询用到的 City、Color 和 Sales Amount 字段从元数据拖曳到数据窗口中，系统将根据拖曳出的列自动生成表格，如图 18.41 所示。

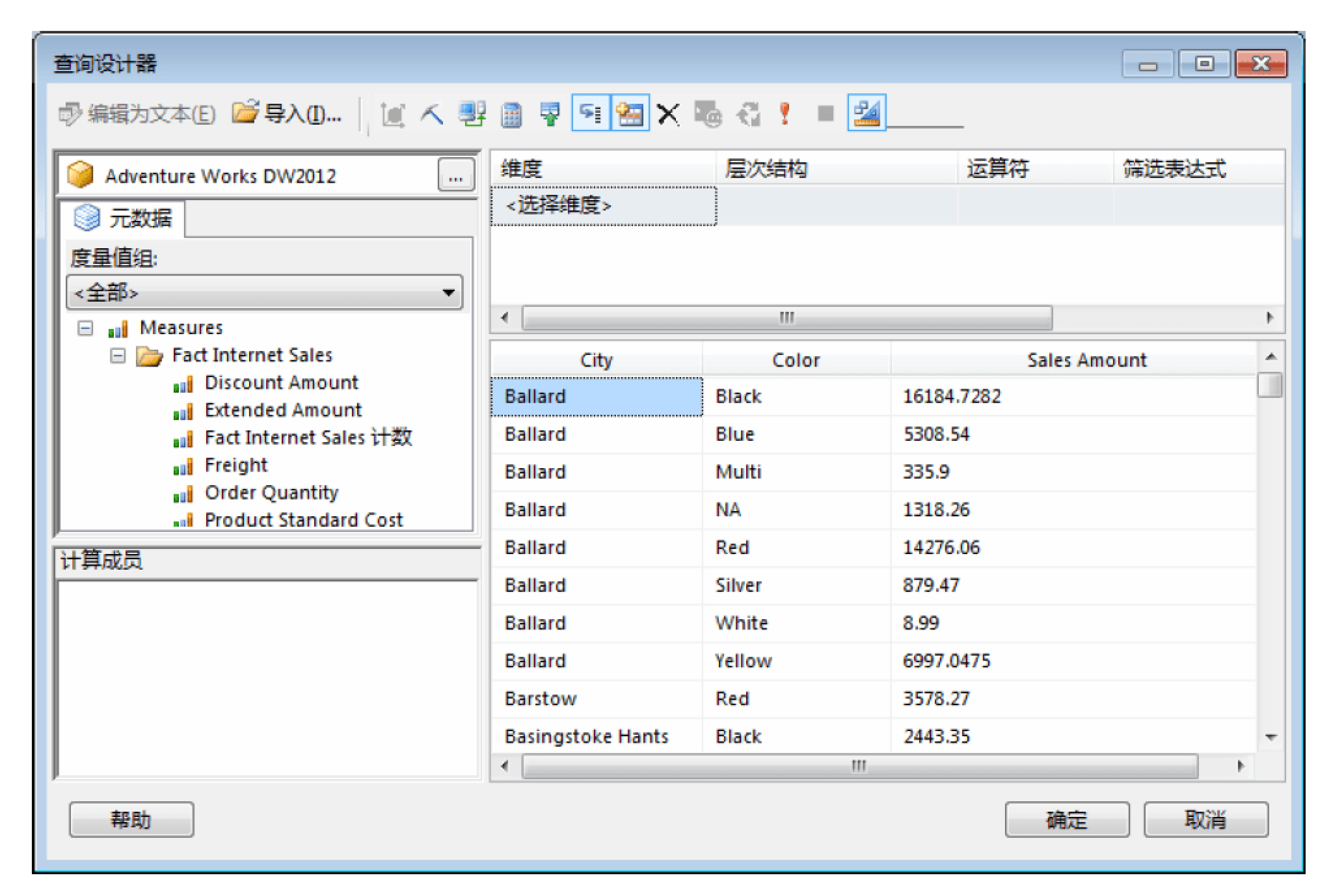


图 18.41 “查询设计器”对话框

（6）单击“确定”按钮回到报表向导，然后单击“下一步”按钮进入报表类型选择界面。这里由于是从 2 个维度分析数据，所以选择矩阵类型。

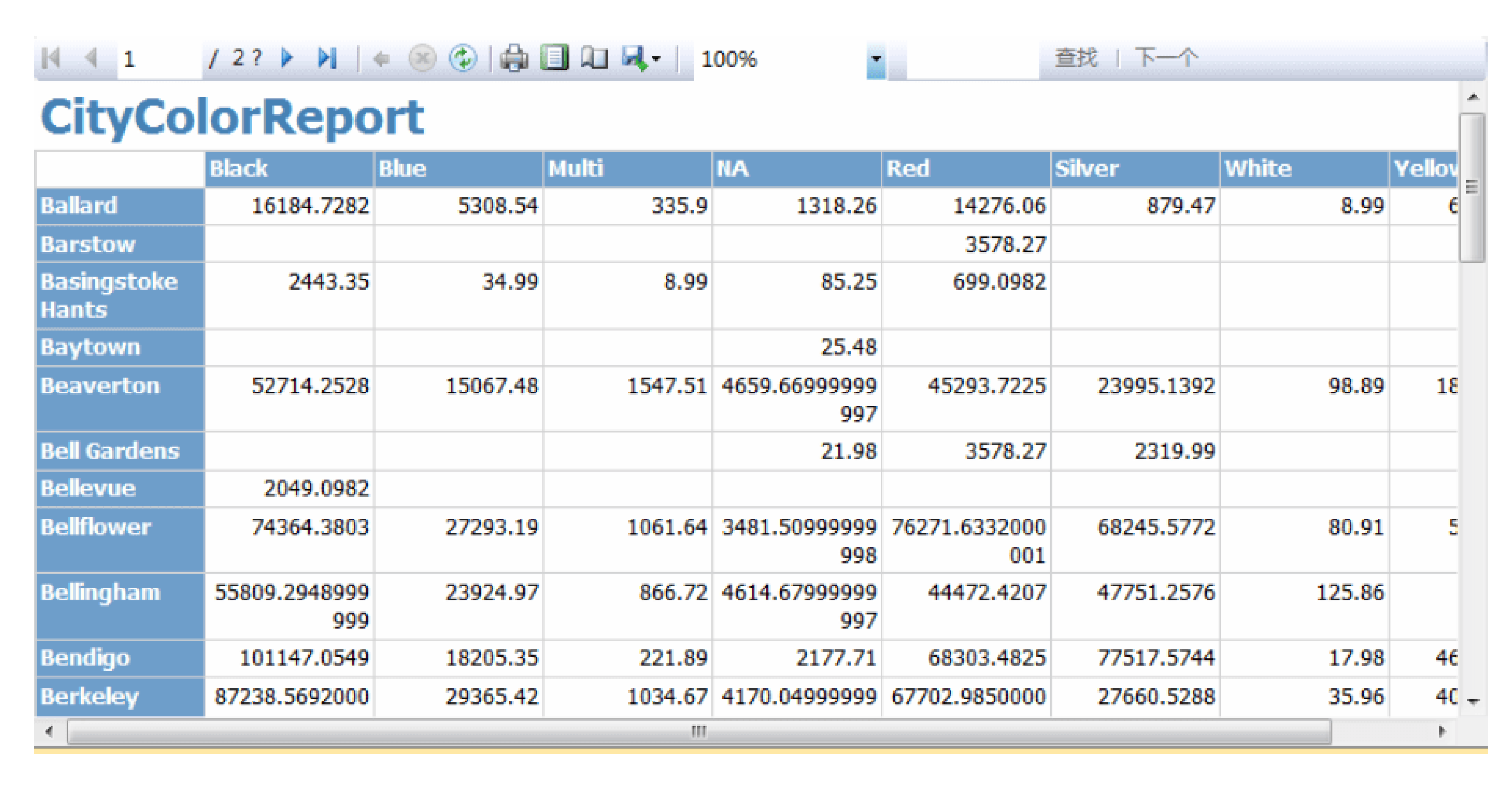
（7）单击“下一步”按钮，进入报表设计矩阵界面，将 Color 添加到列，City 添加到行，Sales_Amount 添加到详细信息中，如图 18.42 所示。



图 18.42　设计矩阵

（8）单击“下一步”按钮，进入报表样式选择界面，随便选择一个样式都可以，然后单击“下一步”按钮，命名报表为 CityColorReport，单击“完成”按钮即可完成该报表的创建。

（9）在报表设计界面，切换到“预览”选项卡可以看到当前报表的预览效果，如图 18.43 所示。



CityColorReport

	Black	Blue	Multi	NA	Red	Silver	White	Yellov
Ballard	16184.7282	5308.54	335.9	1318.26	14276.06	879.47	8.99	6
Barstow					3578.27			
Basingstoke Hants	2443.35	34.99	8.99	85.25	699.0982			
Baytown				25.48				
Beaverton	52714.2528	15067.48	1547.51	4659.66999999997	45293.7225	23995.1392	98.89	18
Bell Gardens				21.98	3578.27	2319.99		
Bellevue	2049.0982							
Bellflower	74364.3803	27293.19	1061.64	3481.50999999998	76271.6332000001	68245.5772	80.91	5
Bellingham	55809.2948999999	23924.97	866.72	4614.67999999997	44472.4207	47751.2576	125.86	
Bendigo	101147.0549	18205.35	221.89	2177.71	68303.4825	77517.5744	17.98	46
Berkeley	87238.5692000	29365.42	1034.67	4170.04999999	67702.9850000	27660.5288	35.96	40

图 18.43　报表预览

用同样的方法可以添加更多的字段到报表中，同时也可以自己定义样式，设计出更复杂、美观的报表。

18.4.3　报表发布

报表在制作完成后可以通过两种方式发布到报表服务器上，一种是通过 VS 2010 的发布功能，另外一种是通过浏览器上传报表文件到报表服务器，在浏览器中设置和管理报表。

在 VS 2010 中发布报表的操作相对简单，与发布 SSAS 相似，通过配置 SSRS 项目的属性，指定发布服务器即可。例如要将创建的报表项目 SSRS1 发布到本机报表服务器上，则右击 SSRS1 项目，在弹出的快捷菜单中选择“属性”选项，在弹出的“属性”对话框中的 TargetServerURL 文本框中，输入报表服务器的虚拟目录，例如 http://servername/reportserver（这是报表服务器的虚拟目录，而不是报表管理器的虚拟目录）。这里是本地报表服务器，所以将其设置为 http://localhost/reportserver。在 StartItem 中，选择报表设计的第一个报表 CityColorReport.rdl。单击“确定”按钮并保存报表项目。接下来右击 SSRS1 项目，在弹出的快捷菜单中选择“部署”选项，系统将会把该项目发布到指定的报表服务器上。

报表服务还提供了 Web 方式的管理界面用于报表服务的管理。默认情况下，报表服务的管理地址是 http://servername/Reports。

注意：如果不清楚报表服务的管理地址，可以在报表服务器上运行 inetmgr 命令启动 IIS 管理器，查看报表服务管理地址的路径和端口。

这里使用的是本机报表服务器，所以在浏览器中输入 http://localhost/Reports，将可以看到报表服务器中的所有报表。如图 18.44 所示为通过 VS 2010 发布 SSRS1 报表项目后的管理界面。在 Web 报表管理器中发布报表的操作如下。

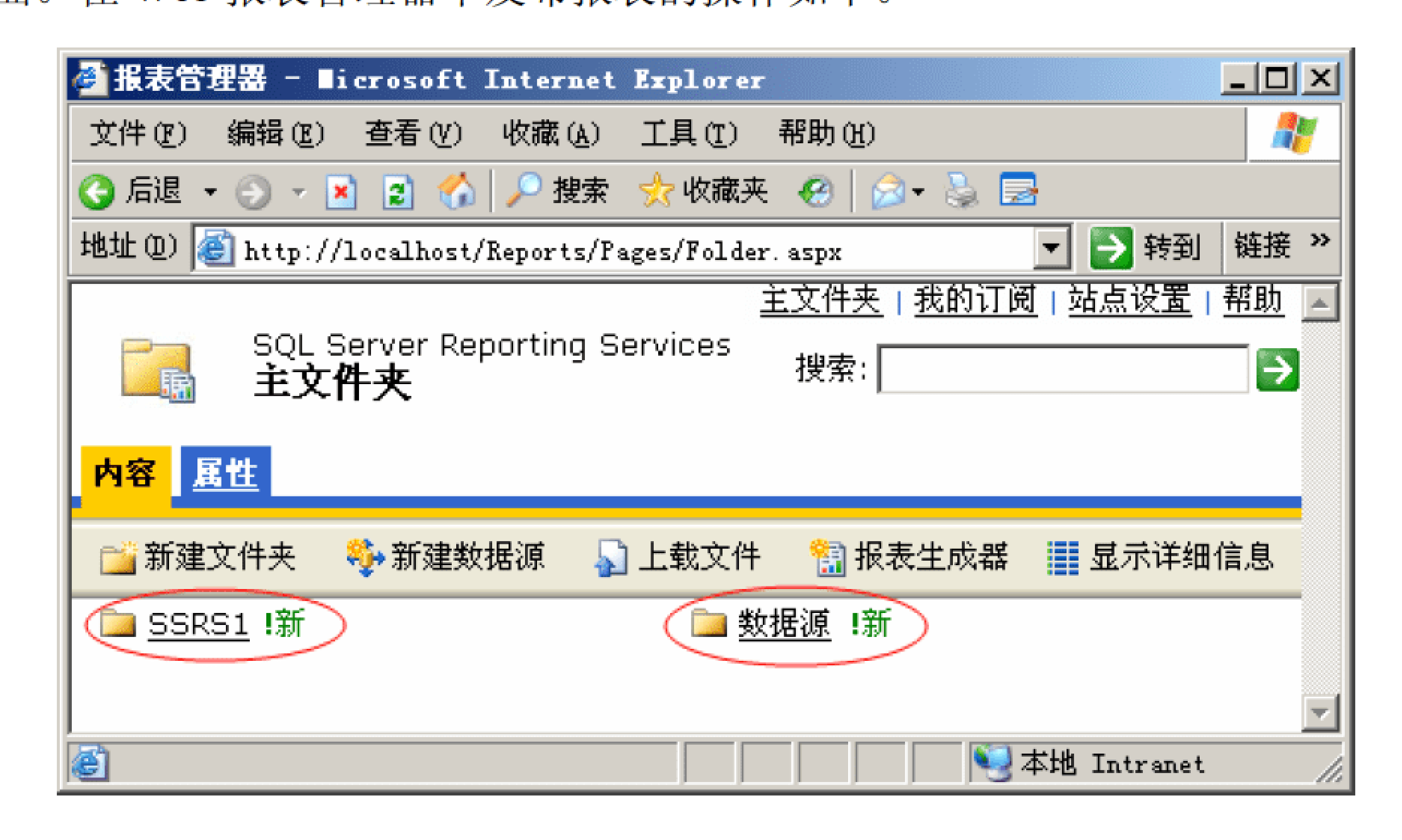


图 18.44　报表管理器

（1）单击“新建数据源”链接，进入新建数据源界面，设置方法与 SSRS 项目中新建数据源相似，在报表管理器中新建连接到 SSAS1 的数据源 SSAS1DataSource。

（2）单击“上传文件”链接，将 VS 2010 中设计好的 CityColorReport.rdl 文件上传到服务器，并命名为 CityColorReport。

（3）上传成功后单击 CityColorReport 链接，进入报表的预览界面，由于该报表的数据源在报表管理器中不存在，所以系统抛出异常信息“报表服务器无法处理该报表。数据源连接信息已删除。”

（4）单击“属性”链接进入 CityColorReport 报表的属性设置界面，在“数据源”选项卡中单击“浏览”按钮，将该报表的数据源设置为前面创建的 SSAS1DataSource，如图 18.45 所示。

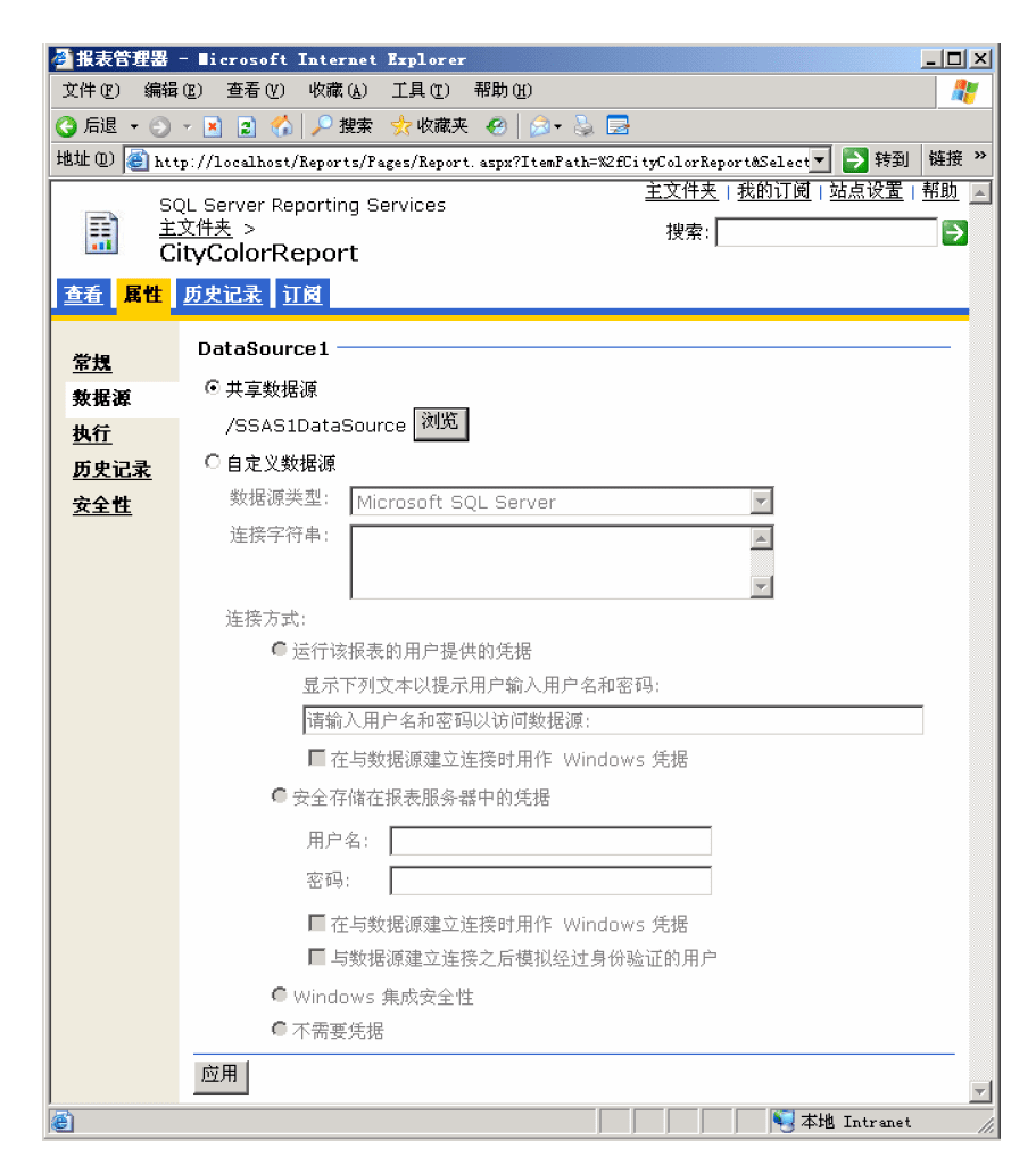


图 18.45　修改报表的数据源

（5）单击“应用”按钮保存对数据源的修改，然后再选择“查看”选项卡，将可以正常查看服务器中的报表，如图 18.46 所示。

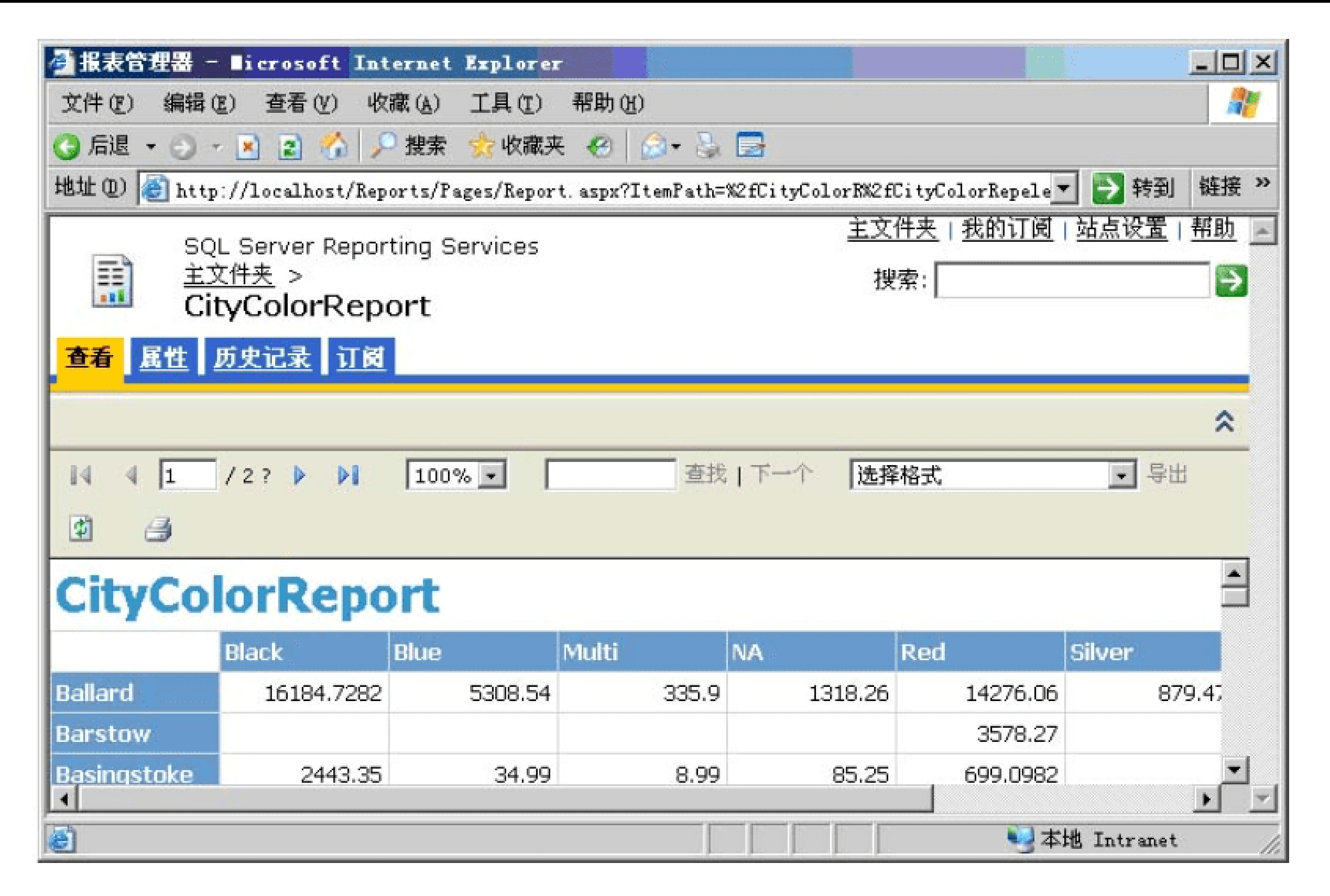


图 18.46　查看报表内容

18.4.4　报表展示

报表发布到报表服务器后，便可通过 IIS 访问报表中的内容。通过访问 http://servername/ReportServer/Pages/ReportViewer.aspx 地址，可以查看报表服务器中的报表。例如要查看通过 Web 报表管理器上传的报表 CityColorReport，则只需要在浏览器中输入 http://localhost/ReportServer/Pages/ReportViewer.aspx?%2fCityColorReport&rs:Command=Render。

但是这种通过直接访问报表服务器查看报表的方式既不安全也不方便管理，报表服务提供了报表查看工具的支持，例如要创建一个 Windows 应用程序，用于显示报表，.NET Framework 提供了专门显示报表的控件 ReportViewer。

在 VS 2010 中使用 C#创建一个 Windows 应用程序，然后从工具箱中将一个 ReportViewer 控件拖曳到窗体中，之后再添加 2 个文本框和一个按钮，在单击按钮时触发单击事件，事件中为 ReportViewer 设置报表路径，具体程序如代码 18.1 所示。

代码 18.1　为 ReportViewer 赋予报表服务属性

```
private void button1 Click(object sender, EventArgs e)
{
   this.reportViewer1.ServerReport.ReportPath = textBox1.Text;
   this.reportViewer1.ServerReport.ReportServerUrl =new Uri
   ( textBox2.Text);
}
```

编译运行该程序，然后输入报表服务器地址和报表地址，单击“查看报表”按钮，刷新报表即可在 Windows 程序中看到生成的报表，如图 18.47 所示。

注意：报表路径必须以正斜杠（/）开头。路径不能包括报表 URL 参数。路径由报表服务器文件夹命名空间中的文件夹和报表名称组成。

另外，在 ASP.NET 中也有 ReportViewer 控件，使用方法与 Windows 下的 ReportViewer 控件的使用方法相同，这里不再举例。

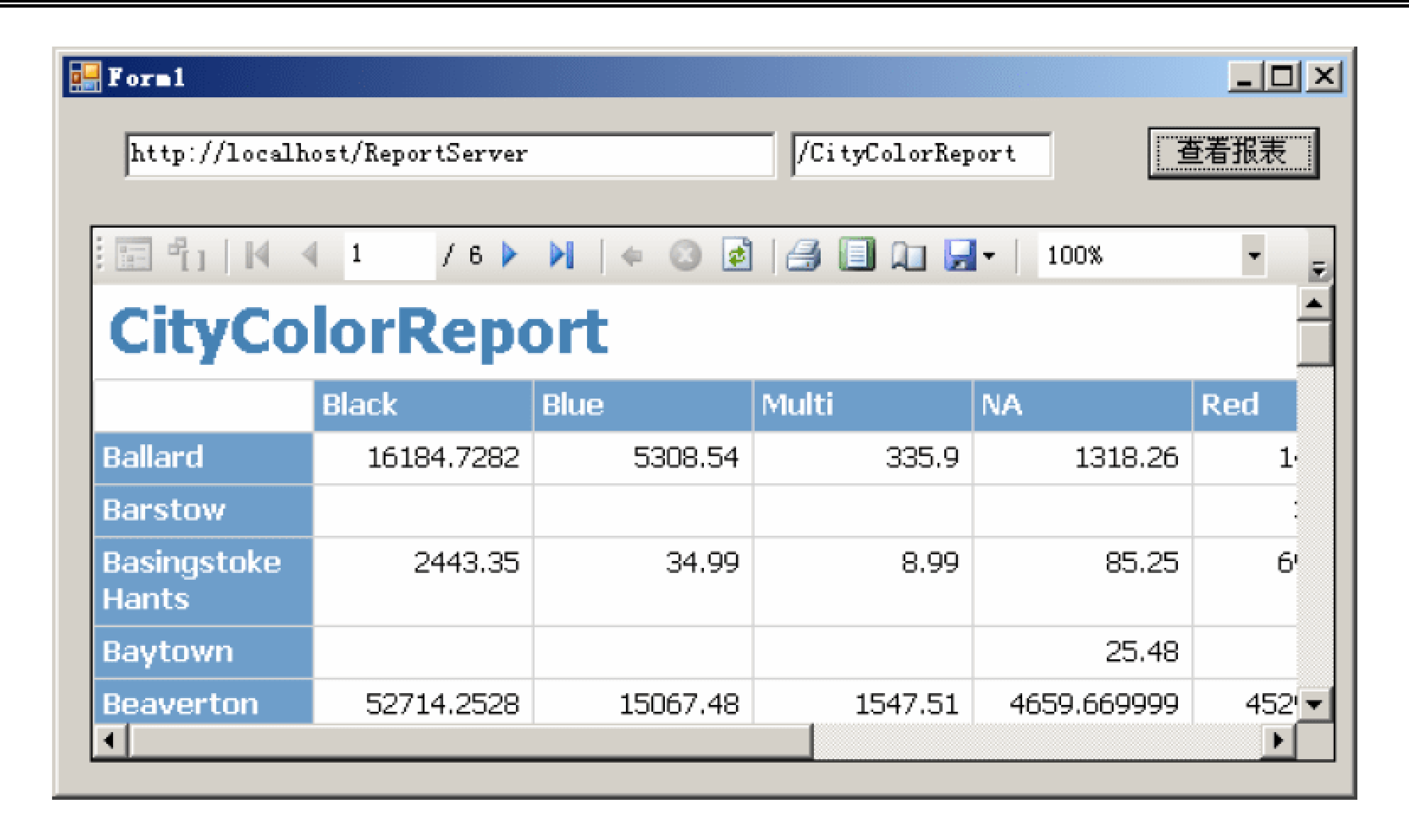


图 18.47　Windows 程序中使用报表服务

18.5　小　　结

本章通过简单的实例主要讲解了 SQL Server 2012 在商务智能上的应用。SQL Server 2012 中的商务智能分为集成服务、分析服务和报表服务 3 大模块。集成服务平台可以生成高性能数据集成解决方案，其中包括为数据仓库提取、转换和加载（ETL）包。分析服务为商务智能应用程序提供联机分析处理（OLAP）和数据挖掘功能。报表服务是基于 Web 的，集创建、管理、分发于一体的报表平台。

第 4 篇　数据库性能优化

第 19 章　数据存储与索引

SQL Server 中数据的存储方式和索引的设置，关系到数据查询和更改时的性能，本章将主要讲解数据的存储方式和数据库索引。在讲解存储方式时，会遇到一些新的概念，如区、页等，这些概念比较抽象，读者阅读完本章会有不小的收获。

19.1　数据库对象分配

所有持久数据库对象最终都将以一定的格式保存到硬盘上，在查询和修改数据时需要对硬盘中的数据进行读写，数据库对象的内在分配势必影响读写数据的性能，本节将主要讲解数据库对象的分配。

19.1.1　对象的存储

在 SQL Server 中最基本的单位是页，1 页由 8KB 组成。数据页是用于存储用户数据的页，数据页有 3 种不同的类型：

- ❑ IN_ROW_DATA，行内数据用于存储堆分区或索引分区。
- ❑ ROW_OVERFLOW_DATA，行溢出数据用于存储超过 8060B 行大小限制的 varchar、nvarchar、varbinary 或 sql_variant 列中存储的可变长度数据。
- ❑ LOB_DATA，大对象数据用于存储大型对象（LOB）数据类型，例如 XML、varbinary(max)和 varchar(max)。

数据页大小固定为 8KB，由页头、数据行和行偏移矩阵组成，如图 19.1 所示。页头大小为 96B，其中标识了该页的编号、上一页的编号、下一页的编号、页类型、该页所属对象 ID、该页空闲字节数等信息。

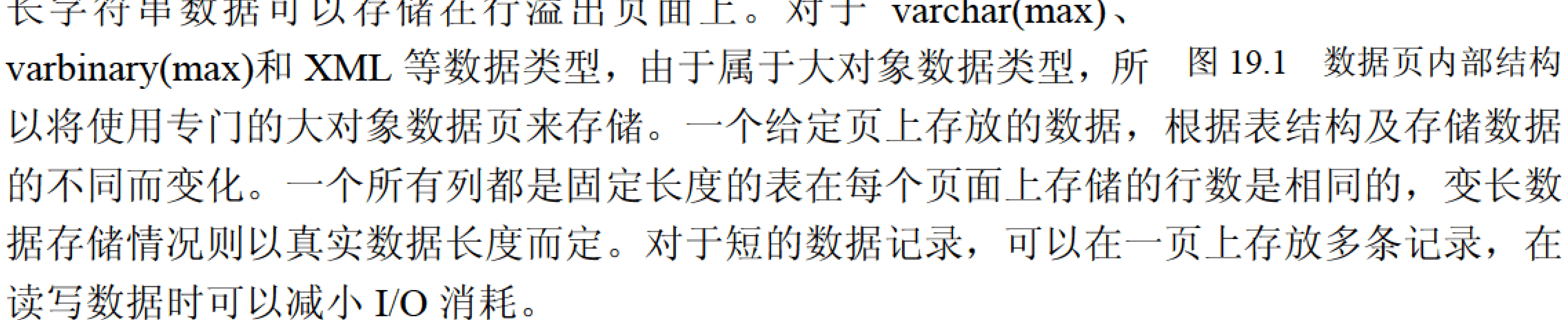


图 19.1　数据页内部结构

对于行内数据，单个数据行最大为 8060B，表中的记录都存储在数据行中，如果记录的总大小超过最大值 8060B 时，表中的变长字符串数据可以存储在行溢出页面上。对于 varchar(max)、varbinary(max)和 XML 等数据类型，由于属于大对象数据类型，所以将使用专门的大对象数据页来存储。一个给定页上存放的数据，根据表结构及存储数据的不同而变化。一个所有列都是固定长度的表在每个页面上存储的行数是相同的，变长数据存储情况则以真实数据长度而定。对于短的数据记录，可以在一页上存放多条记录，在读写数据时可以减小 I/O 消耗。

行偏移矩阵是由 2B 项组成的块。行偏移矩阵用于表示数据记录在数据行中的偏移量，

每条记录在该矩阵中对应一个 2B 的项。偏移矩阵表示了记录在页面上的逻辑顺序，而且是倒序排列，也就是说最后的 2B 表示第 1 行记录的偏移量。记录在数据行中的顺序并不一定是按照聚集索引键值顺序来排列的。

除了数据页以外，还有用于存储索引的索引页，存储大对象的大对象数据页等，具体类型见表 19.1。

表 19.1　页类型

页　类　型	说　　明	内　　容
Data	数据页	当text in row设置为ON时，包含除text、 ntext、image、nvarchar(max)、varchar(max)、varbinary(max) 和 xml 数据之外的所有数据的数据行
Index	索引页	索引条目
Text/Image	大对象页	大型对象数据类型：text、ntext、image、nvarchar(max)、varchar(max)、varbinary(max)和xml数据。数据行超过8KB时为可变长度数据类型列：varchar、nvarchar、varbinary 和 sql_variant
Global Allocation Map Shared Global Allocation Map	全局分配映射页 共享全局分配映射页	有关区是否分配的信息
Page Free Space	页面空间页	有关页分配和页的可用空间的信息
Index Allocation Map	索引分配映射页	有关每个分配单元中表或索引所使用的区的信息
Bulk Changed Map	大容量修改映射页	有关每个分配单元中自最后一条BACKUP LOG语句之后的大容量操作所修改的区的信息
Differential Changed Map	差异映射页	有关每个分配单元中自最后一条BACKUP DATABASE语句之后更改的区的信息

表的数据行在数据页中存储时，先是存储所有固定长度的列，接下来存储所有变长的列。除了记录行数据外，还记录状态位、数据长度、列偏移矩阵等信息。

19.1.2　区-管理空间的基本单位

SQL Server 中每 8 个连续的页组成 1 个区（Extent），区是管理空间的基本单位。为了使空间分配更有效，SQL Server 不会将所有区分配给包含少量数据的表。SQL Server 包含两种类型的区：

- ❑ 统一区，为单个对象所有。区中的所有 8 页只能由所属对象使用。
- ❑ 混合区，最多可由 8 个对象共享。区中 8 页的每页可由不同的对象所有。

由于页单位太小，并且用来管理区分配情况及跟踪可用空间的 SQL Server 数据结构，相对而言比较简单，因此 SQL Server 使用以下两种类型的分配映射表来记录区的分配。

- ❑ 全局分配映射表（GAM），GAM 页记录已分配的区。每个 GAM 包含 64 000 个区，相当于近 4GB 的数据。GAM 用一个 Bit 位来表示所涵盖区间内的每个区的状态。如果位为 1，则区可用；如果位为 0，则区已分配。
- ❑ 共享全局分配映射表（SGAM），SGAM 页记录当前用作混合区且至少有一个未使用的页的区。每个 SGAM 包含 64 000 个区，相当于近 4GB 的数据。SGAM 用一个 Bit 位来表示所涵盖区间内的每个区的状态。如果位为 1，则区用作混合区且

有可用页；如果位为 0，则区未用作混合区，或者虽然用作混合区但其所有页均在使用中。

除了使用 GAM 和 SGAM 表示区的分配情况外，还可使用页可用空间（PFS）记录每页的分配状态、是否已分配单个页及每页的可用空间量。PFS 在每页都有一个字节，记录该页是否已分配。如果已分配，则记录该页是为空，还是已满 1%～50%、51%～80%、81%～95%或 96%～100%。

在数据物理文件中，PFS 页是文件头页之后的第一页（页码为 1），接着是 GAM 页（页码为 2），然后是 SGAM 页（页码为 3）。第一个 PFS 页之后是一个大小约为 8000 页的 PFS 页。在第二页的第一个 GAM 页之后还有另一个 GAM 页（包含 64 000 个区），在第三页的第一个 SGAM 页之后也有另一个 SGAM 页（包含 64 000 个区）。

索引分配映射（IAM）页将映射分配单元使用的数据库文件中 4GB 部分中的区。当 SQL Server 数据库引擎必须在当前页中插入新行，而当前页中没有可用空间时，它将使用 IAM 和 PFS 页查找要将该行分配到的页，或者（对于堆或 Text/Image 页）查找具有足够空间容纳该行的页。数据库引擎使用 IAM 页查找分配给单元的区。对于每个区，数据库引擎将搜索 PFS 页，以查看是否有可用的页。每个 IAM 和 PFS 页覆盖大量数据页，因此一个数据库内只有很少的 IAM 和 PFS 页。这意味着 IAM 和 PFS 页通常位于内存中的 SQL Server 缓冲池中，所以能够很快找到它们。

与 GAM 和 SGAM 类似，SQL Server 还提供了如下两种类型的跟踪映射表来记录区的修改情况：

- 差异更改映射表（DCM）：DCM 页中使用一个 Bit 位来表示一个区是否在上次完整备份后被修改过，如果修改过为 1，未修改过则为 0。在进行差异备份时系统将直接读取 DCM 页中的内容，将标记为 1 的区进行备份。如果执行完整备份，则所有 DCM 页中的位被置 0。
- 大容量更改映射表（BCM）：BCM 页中跟踪记录了从上次日志备份后被大容量日志记录操作修改的区。如果上次执行日志备份后，某区被有日志记录的大容量复制操作修改，对应的 Bit 位为 1，未被修改则为 0。只有在大容量恢复模式下 BCM 页才有用，当执行日志备份时，系统将根据 BCM 页找到已修改的区，将这些区包括在日志备份中，以便在数据库还原时恢复大容量日志记录操作。

DCM 和 BCM 页都可以表示 640 000 个区，在物理文件中，DCM 和 BCM 页在 GAM 和 SGAM 页之后。

19.2　索　　引

数据库中存放了大量的数据，为了能够快速地查找需要的数据，SQL Server 使用了索引来组织和查找，从而提高了数据访问速度。

19.2.1　索引简介

除了表之外，索引是另一个重要的自定义数据结构，使用索引可以加快从表或视图中检索行的速度。索引包含由表或视图中的一列或多列生成的键。这些键以一个 B 树的数据

结构进行存储，以便 SQL Server 可以快速有效地查找与键值关联的行。

索引使得数据以一种特定的方式组织起来，从而使数据访问时获得最佳的性能。SQL Server 中索引分为聚集索引和非聚集索引。一个表中最多只能有一个聚集索引，但是却可以有多个非聚集索引。没有建立聚集索引的表叫做堆（Heap）。在进行数据查找时，如果表中没有可以利用的索引，则查询优化器必须扫描表，对表进行扫描会有许多磁盘 I/O 操作，并占用大量资源。但并不是只要建立了索引的表在查询时就一定会使用索引，如果查询表达式不符合使用索引的规范或者返回的结果集的行数占表总行数的百分比较高，则系统仍然会扫描表进行查找。

说明： 这里介绍的索引都是指一般数据类型的索引，不包括 XML 类型和空间数据类型的索引，这两个索引都将在相应的章节中进行介绍。

索引可以是唯一的（无论是聚集索引还是非聚集索引），表示其中的数据并没有重复，也可以不是唯一的，即多行可以共享同一键值。

SQL Server 中的索引使用标准的 B 树来存储索引信息。B 树又叫平衡树，图 19.2 为 B 树的结构示意图，系统通过查找索引中的一个关键值来提供对数据的快速访问。B 树以相似的键将记录聚合在一起。B 树的一个核心就是维护树的平衡，树被实时维护，这样通过向下遍历这棵树以找到一个数值，并定位一个特定记录只需要经过几次页面访问即可。由于要维护树的平衡，所以在对数据进行插入、修改和删除时，需要花费一定的资源来修改 B 树。

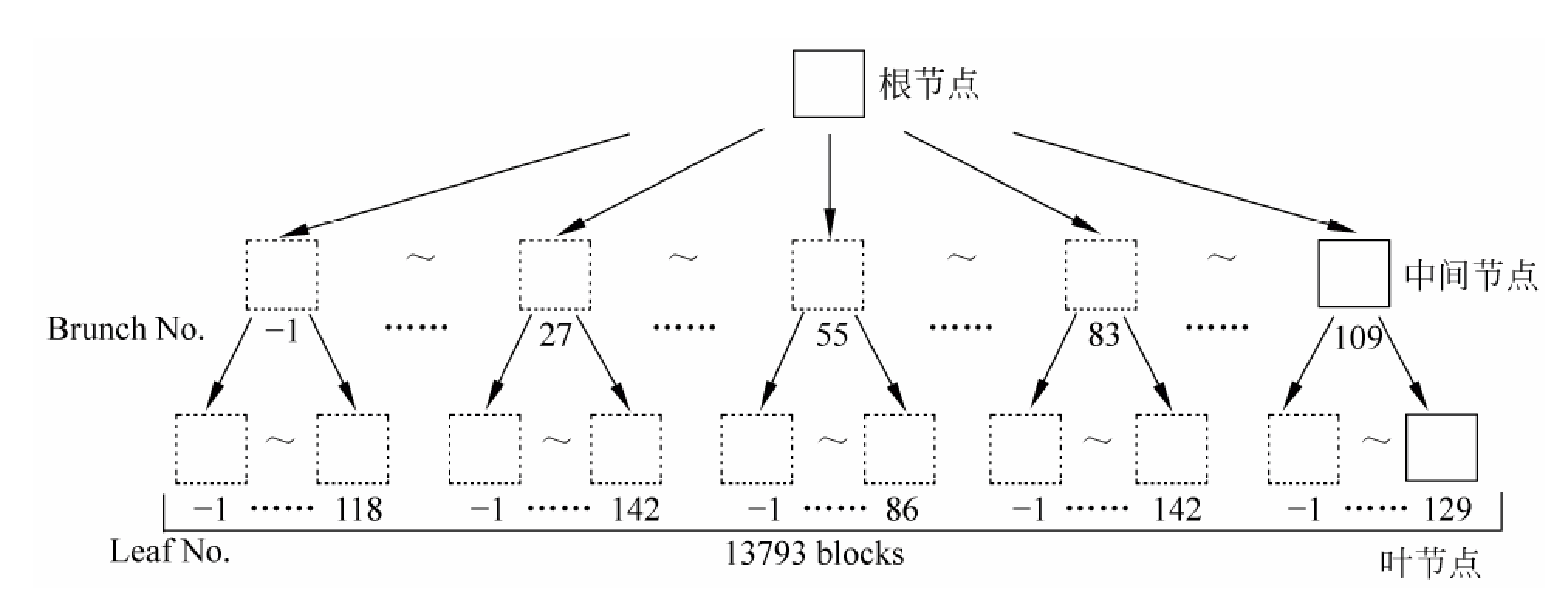


图 19.2　B 树结构

索引中具有唯一的作为遍历起点的根分页，可能存在中间索引层次及底层叶子分页。由于使用 B 树结构，因此使用索引查找过程中从根出发可以很快找到需要的叶子分页。索引的中间层次是根据表的行数及索引行的大小而变化的。若使用较长的键值来创建索引，一个页中就只能容纳较少的条目，因此该树就需要更多的页，在查询时将会使用更多的 I/O 操作，所以在创建表的列时，特别是用于索引的列时，应该尽量使用占用空间较短的数据类型。对于任何索引，无论是聚集的还是非聚集的，叶级别都是按照键的顺序由所有的键值组成。

19.2.2　聚集索引

聚集索引确定了数据存储的顺序，表内的数据存储基于聚集索引键值在表内排序。每

个表只能有一个聚集索引，因为数据行本身只能按一个顺序存储。

聚集索引就像是字典的编排方式，所有单词按照字母顺序排列，一本字典中单词的排列方式只有一种，通过字母排序的好处是在查找某个单词时不需要去查看前面的目录，只需要任意翻开一页，然后根据当页的单词就能够判断要查找的单词是在此页的前面还是后面。

说明：默认情况下，在创建表时若没有指定聚集索引在哪个列上，则系统会为表的主键创建唯一的聚集索引。

SQL Server 中使用 index_id=1 来标识索引类型为聚集索引。SQL Server 在使用聚集索引查找数据时，从 B 树的根开始在索引中向下移动以查找与某个聚集索引键对应的行。由于 B 树中的数据按照一定的顺序排序，所以就像是做二分查找一样，系统将在索引中移动以查找该范围的起始键值，然后用向前或向后指针在数据页中进行扫描，从而快速地确定数据所在的范围。为了查找数据页链的首页，SQL Server 将从索引的根节点沿最左边的指针进行扫描。如图 19.3 所示为聚集索引在单个数据分区中的结构。

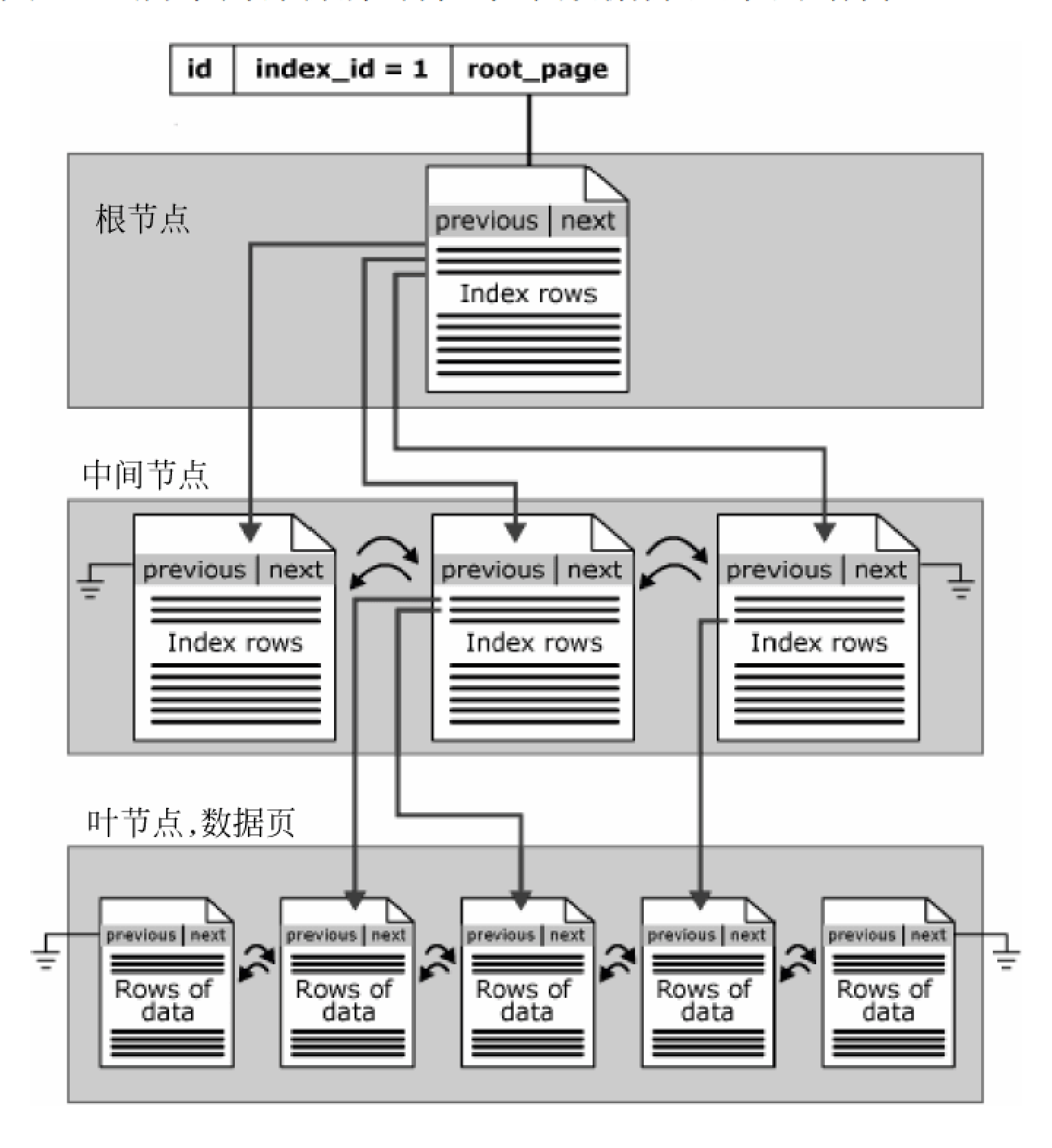


图 19.3　聚集索引结构

一般情况下，定义聚集索引键时使用的列越少越好。可考虑具有下列一个或多个属性的列：

- 唯一或包含许多不重复的值。
- 按顺序被访问。
- 由于保证了列在表中是唯一的，所以定义为 IDENTITY。
- 经常用于对表中检索到的数据进行排序。

对于频繁修改的列和数据长度较长的列，则不适合建立聚集索引。

19.2.3　非聚集索引

非聚集索引与聚集索引有一个相似的 B 树索引结构。不同的是，非聚集索引不影响数据行的顺序。在 B 树结构中，叶级别只包含索引键列和包含列的值，而不像聚集索引一样是数据页。每个索引行包含非聚集键值、行定位符（row identifier，RID）和任意包含列或非键列。定位符指向包含键值的数据行。

非聚集索引就像是字典前面的索引目录，在新华字典中既可以通过拼音在目录中查找汉字的页数，也可以通过偏旁笔画来查找汉字的页数。一个字典中可以建立多个索引目录，索引目录中只给出了要查找的字的页数，并没有给出字的含义、用法。与之对应的，一个表中也可以建立多个非聚集索引，非聚集索引中包含的行定位符相当于字典中的页数，通过行定位符再找到具体行中的所有数据。字典中的索引目录独立于正文存在，并不影响正文的内容和顺序，同样，非聚集索引也是独立于表内容页的，并不影响内容页的顺序。

SQL Server 中使用 index_id>1 来标识索引类型为非聚集索引。每个表中最多允许包含 249 个非聚集索引。非聚集索引与聚集索引使用的都是 B 树结构，但是其叶层由索引页而不是数据页组成。查找的过程中先是使用与查找聚集索引相同的方法查找 B 树，在查找到叶层后，如果所需的数据并没有完全在页层中记录，还要根据页中的行定位符在数据页中查找对应的数据。整个非聚集索引的结构如图 19.4 所示。

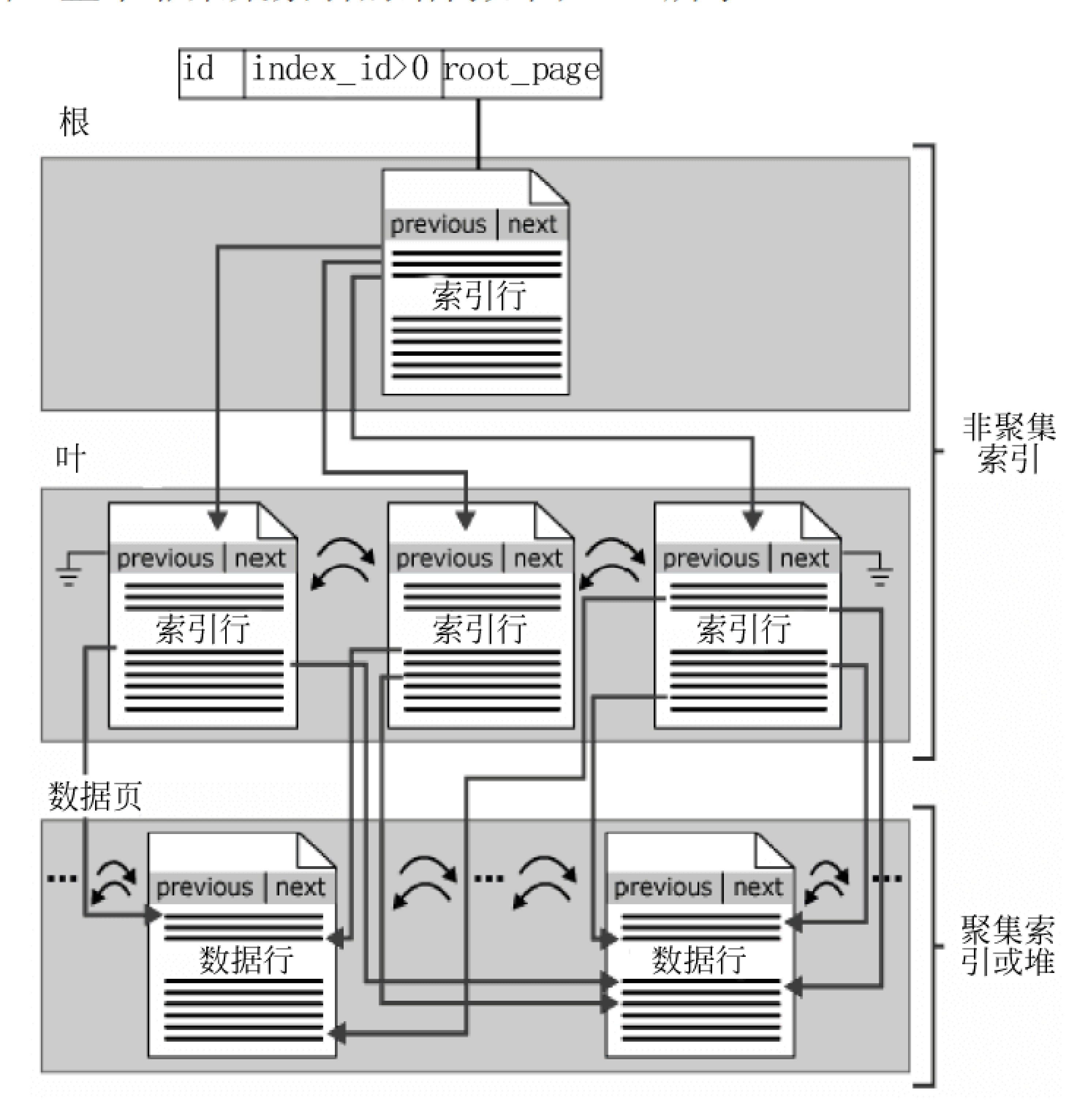


图 19.4　非聚集索引结构

SQL Server 从 2005 版本起具有可以将包含列（称为非键列）添加到索引的叶级和扩

展非聚集索引的功能。键列存储在非聚集索引的所有级别，而非键列仅存储在叶级别。通过包含非键列，可以创建覆盖更多查询的非聚集索引。非键列具有下列优点：

- 可以是不允许作为索引键列的数据类型。
- 在计算索引键列数或大小时，数据库引擎不考虑非键列。

当查询中的所有列都作为键列或非键列包含在索引中时，带有包含性非键列的索引可以显著提高查询性能。这样可以实现性能提升，因为查询优化器可以在索引中找到所有列值，而无须访问表或聚集索引数据，从而减少磁盘 I/O 操作。但是包含非键列后会产生以下问题：

- 一页上能容纳的索引行更少。
- 需要更多的磁盘空间来存储索引。
- 索引维护可能会增加对基础表或索引视图执行修改、插入、更新或删除操作所需的时间。

所以并不能盲目地使用包含非键列的非聚集索引，否则会适得其反，降低数据库性能。

另外，在 SQL Server 2008 中添加了一个新特性：筛选索引。筛选索引就是在建立非聚集索引时，使用 WHERE 条件对建立索引的数据进行筛选，只有满足条件的数据才建立索引。由于使用筛选索引后只对表中的部分数据建立索引，所以建立的 B 树可能会比一般的非聚集索引小很多。如果查询的数据都在筛选索引上，由于该索引树较小，所以检索速度也更快。比如一个表中有 1 万条数据，但是其中只有 100 条数据的 A 列不为空，那么可以对 100 条非空数据建立筛选索引，在查询该列时可以提高查询的效率。

在下面的情况下可以考虑使用筛选索引，而不是一般的非聚集索引：

- 仅包含少量非 NULL 值的稀疏列。
- 包含多种类别的数据的异类列。
- 包含多个范围的值（如美元金额、时间和日期）的列。
- 由列值的简单比较逻辑定义的表分区。

在 SQL Server 2012 中，选择性 XML 索引是除了普通 XML 索引之外还可以使用的另外一种 XML 索引类型。它能够改进在 SQL Server 中存储的 XML 数据的查询性能，支持更快速地对较大的 XML 数据工作负荷建立索引，通过减少 XML 索引的存储成本提高可伸缩性。

19.2.4　堆

堆是不含聚集索引的表。为了管理方便，SQL Server 认为堆是一种特殊的索引，使用 index_id = 0 来标识。堆的结构正如其名，就是将数据页没有任何规律和顺序地摆放在一起，就像散乱的书页一样，知道了当前页是第 2 章，但是由于没有顺序，第 1 章是在该书页的前面还是后面却无法知道。

堆内的数据页和行没有任何特定的顺序，也不链接在一起。数据页之间唯一的逻辑连接是记录在 IAM 页内的信息。在堆中查找数据时可以通过扫描 IAM 页，然后对堆进行表扫描或串行读操作来找到容纳该堆的页的扩展盘区。因为 IAM 是按扩展盘区在数据文件内存在的顺序表示它们，所以这意味着串行堆扫描是连续沿每个文件进行的，而不是按照表

中的数据行插入的顺序进行。堆的内部结构如图 19.5 所示。

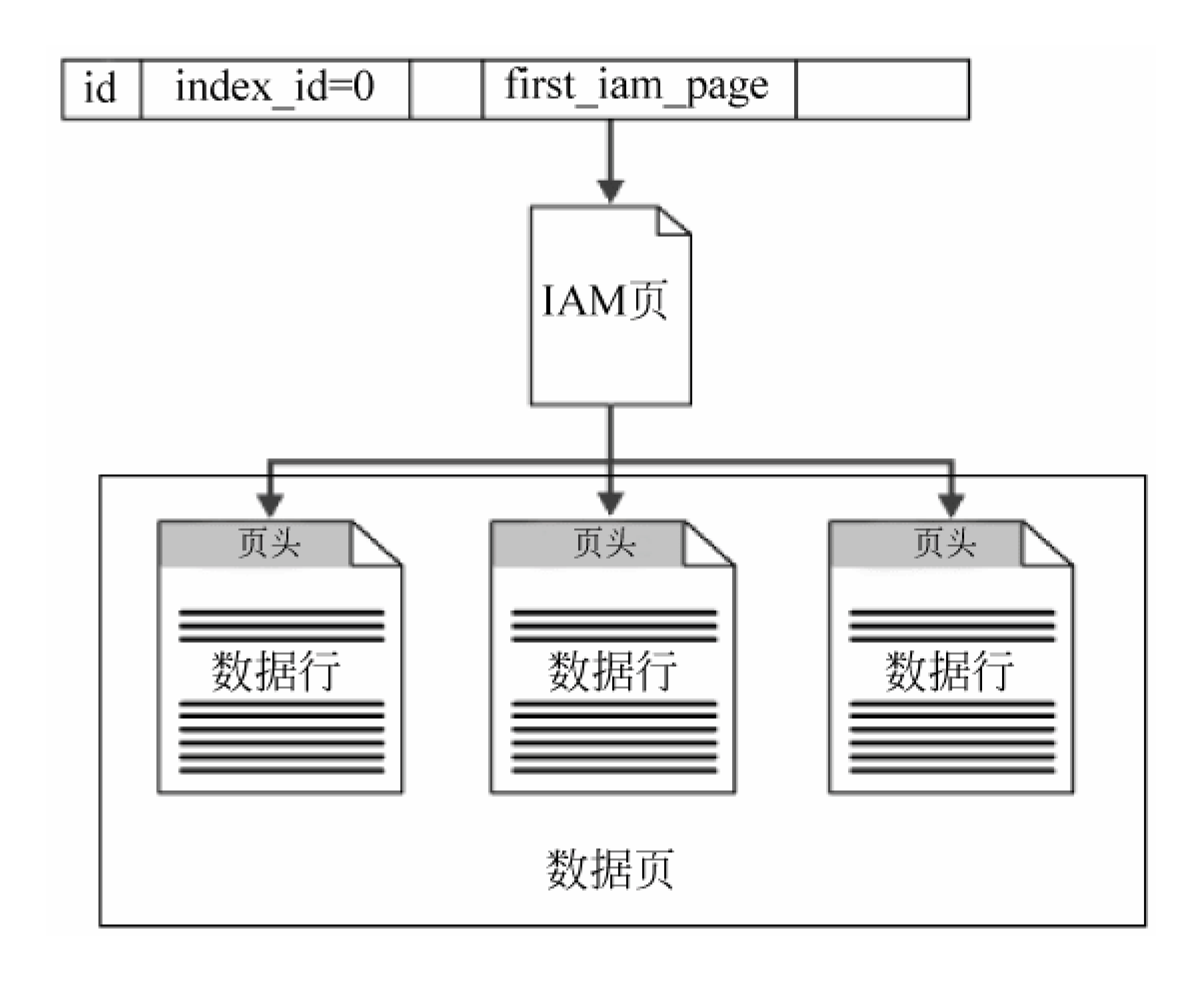


图 19.5　堆的结构

堆虽然没有聚集索引的表，仍然可以在表中为堆创建非聚集索引。由于堆的数据存放无序不便于查找，所以在实际项目中除很特殊的情况外，都不会使用堆。

19.2.5　创建索引

SQL Server 提供了 CREATE INDEX 命令用于创建索引。CREATE INDEX 的语法如代码 19.1 所示。

代码 19.1　CREATE INDEX 语法

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index name
    ON [ database name. [ schema name ] . | schema name. ] table or view name
    ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <filter_predicate> ]
    [ WITH ( <relational_index_option> [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name )
         | filegroup_name
         | default
         }
    ]
    [ FILESTREAM ON { filestream filegroup name | partition scheme name |
"NULL" } ]

<relational index option> ::=
{
    PAD INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT IN TEMPDB = { ON | OFF }
  | IGNORE DUP KEY = { ON | OFF }
  | STATISTICS NORECOMPUTE = { ON | OFF }
  | DROP EXISTING = { ON | OFF }
  | ONLINE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
```

```
  | DATA COMPRESSION = { NONE | ROW | PAGE}
    [ ON PARTITIONS ( { <partition number expression> | <range> }
    [ , ...n ] ) ]
}
```

其中最常用的几个参数如下。

- UNIQUE 为表或视图创建唯一索引。唯一索引不允许两行具有相同的索引键值。视图的聚集索引必须唯一。
- CLUSTERED 创建聚集索引，键值的逻辑顺序决定表中对应行的物理顺序。如果没有指定 CLUSTERED，则创建非聚集索引。
- NONCLUSTERED 创建非聚集索引。对于非聚集索引，数据行的物理排序独立于索引排序。
- index_name 为索引的名称。
- column 为索引所基于的一列或多列。一个组合索引键中最多可组合 16 列。组合索引值允许的最大大小为 900B。不能将大型对象数据类型的列指定为索引的键列。
- [ASC|DESC]确定特定索引列的升序或降序排序方向，默认值为 ASC。
- INCLUDE(column[,...n])指定要添加到非聚集索引的叶级别的非键列。非聚集索引可以唯一，也可以不唯一。可包含的非键列的最大数量为 1023 列，最小数量为 1 列。

前面已经讲到，默认情况下，在创建表的主键时将在主键列上创建与主键同名的唯一聚集索引。如果要让系统在创建主键时不将主键列创建为聚集索引，需要在 PRIMARY KEY 关键字后使用 NONCLUSTERED 关键字。例如创建一个学生表，将学号作为该表的主键，但是聚集索引在学生的姓名上，则对应的 SQL 脚本如代码 19.2 所示。

代码 19.2　创建学生表和名字列上的聚集索引

```
CREATE TABLE Student
(
    StuID INT IDENTITY PRIMARY KEY NONCLUSTERED, --指定该主键为非聚集索引
    Name NVARCHAR(10) NOT NULL,
    Birthday DATE NOT NULL,
    Sex BIT NOT NULL DEFAULT(0)
)
GO
CREATE CLUSTERED INDEX CIX_Student_Name --单独创建聚集索引
ON Student(Name)
```

如果要创建唯一索引，需要使用 UNIQUE 关键字。例如创建一个选课表，在其中有自增列作为主键和聚集索引，课程编号与学号的组合应该是唯一的，所以可以在这两列上建立唯一的非聚集索引，具体 SQL 脚本如代码 19.3 所示。

代码 19.3　创建选课表和唯一的非聚集索引

```
CREATE TABLE ChooseCourse
(
    ChooseID INT IDENTITY PRIMARY KEY, --这里是聚集索引
    CourseID INT NOT NULL,
    StuID INT NOT NULL,
```

```
    Status TINYINT NOT NULL
)
CREATE UNIQUE NONCLUSTERED INDEX IX ChooseCourse CourseIDStuID
ON ChooseCourse(CourseID,StuID)
```

使用 INCLUDE 关键字可以创建具有包含列的非聚集索引，例如在学生表的 Birthday 上创建非聚集索引，同时将学生的性别添加到包含列中，则对应的 SQL 脚本如代码 19.4 所示。

代码 19.4　创建有包含列的索引

```
CREATE INDEX IX Student Birthday
ON Student(Birthday)
INCLUDE(Sex)
```

使用 SSMS 创建索引的操作十分简单，在 SSMS 的对象资源管理器中展开需要创建索引的表节点，选择“表”节点下的“索引”节点，在弹出的快捷菜单中选择“新建索引”选项，弹出“新建索引”对话框帮助设置索引，如图 19.6 所示。

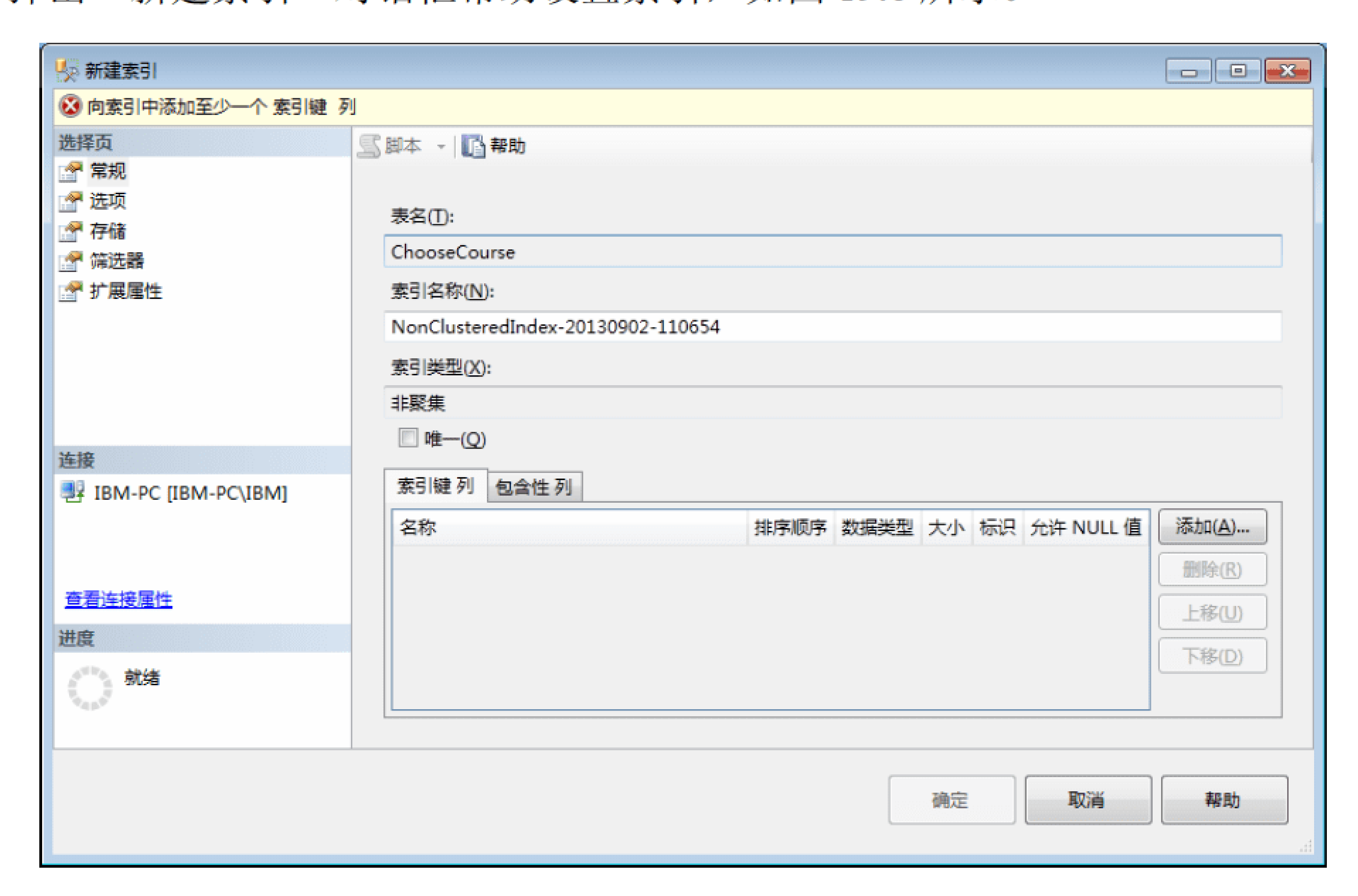


图 19.6　“新建索引”对话框

按照提示输入索引名称，选择索引类型和唯一性，添加索引列，如果有包含列，选择“包含性列”选项添加包含列，设置非键列，最后单击“确定”按钮即可完成索引的创建。

19.2.6　管理索引

索引在创建好后，其内部结构会随着数据的变化而改变，从而造成索引的效率降低，这时就需要重新生成或者重新组织索引。如果由于某种原因不希望使用索引，可以将索引禁用，在需要时再启用。无论是重建索引、重新组织索引或者禁用和启用索引，都是通过 ALTER INDEX 命令实现。ALTER INDEX 命令的语法如代码 19.5 所示。

代码 19.5　ALTER INDEX 命令语法

```
ALTER INDEX { index_name | ALL }
    ON [ database_name. [ schema_name ] . | schema_name. ]table_or_view_name
    { REBUILD
        [ [ WITH ( <rebuild index option> [ ,...n ] ) ]
          | [ PARTITION = partition number
                [ WITH ( <single_partition_rebuild_index_option>
                        [ ,...n ] )
                ]
            ]
        ]
    | DISABLE
    | REORGANIZE
        [ PARTITION = partition number ]
        [ WITH ( LOB COMPACTION = { ON | OFF } ) ]
  | SET ( <set index option> [ ,...n ] )
    }
```

其中几个常用的参数如下。

- index_name 为索引的名称。
- ALL 指定与表或视图相关联的所有索引，而不考虑是什么索引类型。
- table_or_view_name 为与该索引关联的表或视图的名称。
- REBUILD 指定将使用相同的列、索引类型、唯一性属性和排序顺序重新生成索引。
- DISABLE 将索引标记为已禁用，从而不能由数据库引擎使用。
- REORGANIZE 指定将重新组织的索引叶级。

注意：ALTER INDEX 语句不能用于修改索引定义，如添加或删除列，或更改列的顺序。若要修改定义，需要删除原有索引，然后重新使用 CREATE INDEX 或者 CREATE INDEX WITH DROP_EXISTING 命令创建索引。

重新生成索引的过程实际上是先删除原有索引，然后重新创建索引，这将根据指定的或现有的填充因子，设置压缩页来删除碎片、回收磁盘空间，然后对连续页中的索引行重新排序。如果指定 ALL，将删除表中的所有索引，然后在单个事务中重新生成。例如要重新生成选课表的非聚集索引，对应的 SQL 脚本如代码 19.6 所示。

代码 19.6　重建索引

```
ALTER INDEX IX_ChooseCourse_CourseIDStuID
ON dbo.ChooseCourse
REBUILD
```

重新组织索引使用最少系统资源。通过对叶级页以物理方式重新排序，使之与叶节点的从左到右的逻辑顺序相匹配，进而对表和视图中的聚集索引和非聚集索引的叶级进行碎片整理。例如要重新组织学生表上的所有索引，对应的 SQL 脚本如代码 19.7 所示。

代码 19.7　重新组织索引

```
ALTER INDEX ALL
ON Student
REORGANIZE
```

禁用索引可防止用户访问该索引，对于聚集索引，还可防止用户访问基础表数据。索引定义保留在系统目录中。对视图禁用非聚集索引或聚集索引会以物理方式删除索引数据。禁用聚集索引将阻止对数据的访问，但在删除或重新生成索引之前，数据在 B 树中一直保持未维护的状态。例如要通过禁用学生表上的索引来防止用户访问该表，对应的 SQL 脚本如代码 19.8 所示。

代码 19.8　禁用索引

```
ALTER INDEX ALL
ON Student
DISABLE
GO
SELECT * --聚集索引被禁用，查询将抛出异常
FROM Student
```

禁用索引后若需要再恢复，只需要使用 ALTER INDEX REBUILD 语句重新生成即可。

删除索引通过 DROP INDEX 语句实现，例如要删除学生表上的一个索引，对应的 SQL 脚本为：

```
DROP INDEX CIX_StudentName
ON Student
```

注意： DROP INDEX 命令不能用于删除通过定义 PRIMARY KEY 或 UNIQUE 约束创建的索引。若要删除该约束和相应的索引，要使用带有 DROP CONSTRAINT 子句的 ALTER TABLE 命令。

在 SSMS 下管理索引，只需要通过在对象资源管理器中找到该索引，然后右击它，再从弹出的快捷菜单中选择具体操作，如重新生成、重新组织、禁用或删除。

19.3　索 引 选 项

在创建索引时除了提供最基本的索引名、索引类型、索引键列外，还提供了其他高级选项用于设置和维护索引，使得索引的使用更加高效。

19.3.1　填充因子

填充因子（Fill Factor）选项用来设置叶子层级页要放置记录的满溢程度，填充因子确定每个叶级页上要填充数据的空间百分比，以便保留一定百分比的可用空间供以后扩展索引。在 SQL Server 中提供填充因子选项是为了优化索引数据存储和性能。填充因子值是 1～100 之间的百分比值，服务器范围的默认值为 0，表示将完全填充叶级页。

注意： 填充因子值 0 和 100 意义相同。

在 SQL Server 中，索引使用 B 树结构存储，在进行数据更改时（特别是在中间添加行或者删除行）需要实时维护 B 树的结构，以保持树的平衡性。如果向已满的索引页添加新

行，数据库引擎将把大约一半的行移到新页中，以便为该新行腾出空间，这种重组称为页拆分。

页拆分主要是为新记录腾出空间，但是执行页拆分需要进行磁盘 I/O 操作，可能需要花费一定的时间，并且会消耗大量资源。此外，它还可能产生碎片，从而导致 I/O 操作增加。如果经常发生页拆分，可通过使用新的或现有的填充因子值来重新生成索引，从而重新分发数据。如图 19.7 所示为插入一定数据后，查看索引 IX_Student_Birthday 的碎片和填充度情况。

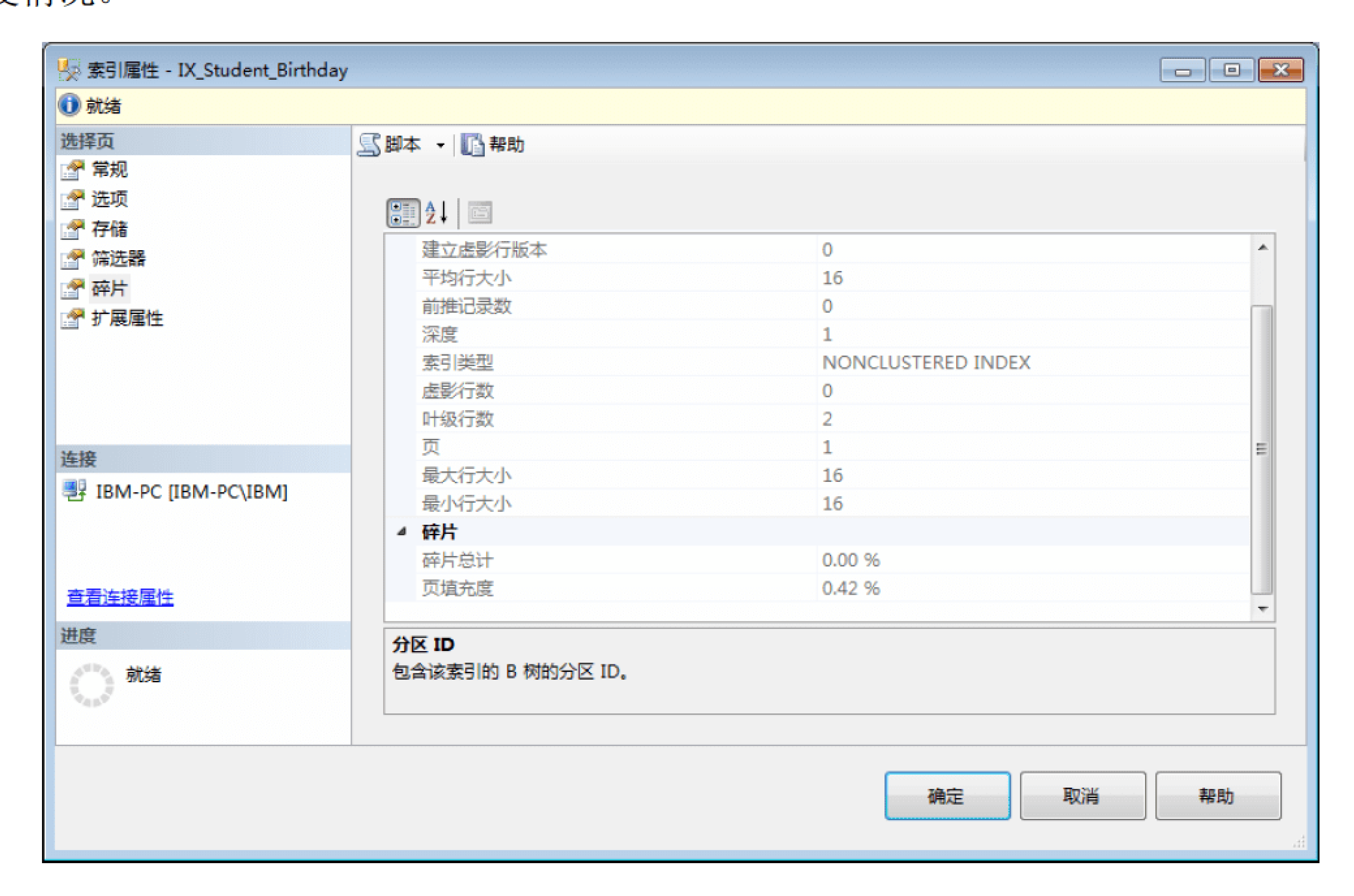


图 19.7　索引的页填充度和碎片

虽然采用较低的填充因子值（非 0）可减少随着索引增长而拆分页的需求，但是一个页上存储的数据将更少，整个索引 B 树将占用更多的存储空间，从而导致更多的 I/O 操作，降低读取性能。

如果新数据在表中均匀分布，则非零填充因子对性能有利。但是，如果所有数据使用了 IDENTITY 或者当前时间作为索引列，那么添加数据时都是添加到表的末尾，则不会填充空的空间。在这种情况下，页拆分将不会导致性能下降，因此应当使用默认填充因子 0，或者指定填充因子 100，以便在叶级进行填充，减少低填充因子情况小的 I/O 操作。

可以使用 CREATE INDEX 或 ALTER INDEX 语句来设置各个索引的填充因子值。使用 ALTER INDEX 语句后，SQL Server 并不会立即将现有索引按照指定的填充因子重排，需要重新组织索引或重建索引才会生效。

如果不希望一个个地设置索引的填充因子，可以通过修改服务器属性来设置全局默认填充因子。修改服务器填充因子通过 sp_configure 设置服务器选项的“fill factor(%)”来实现。例如要修改服务器的填充因子为 80%，则对应的 SQL 脚本如代码 19.9 所示。

代码 19.9　修改服务器填充因子

```
EXEC sys.sp_configure N'show advanced options', N'1'
RECONFIGURE WITH OVERRIDE
```

```
GO
EXEC sys.sp configure N'fill factor (%)', N'80'
GO
RECONFIGURE WITH OVERRIDE
```

在 SSMS 中，也可以通过修改服务器属性中“数据库设置”选项卡的默认索引填充因子来实现。

技巧：最好的索引维护方式是通过维护计划来重建索引，在凌晨时间业务系统空闲时将需要的索引重建，避免由于长时间修改数据造成的索引不连续。

填充因子设置的只是叶子层级的数据填充程度，如果需要让填充因子也应用到索引 B 树的中间层级，则可以使用 PAD_INDEX 选项。该选项将参照填充因子的设置来保留中间层级的数据填充度，其本身并不设置百分率。

19.3.2　联机索引操作

默认在建立索引时，为了避免其他用户同时修改到相关数据，系统会对相关的表设置共享锁定（关于锁的信息可以参见第 21 章）。如果在创建索引时带上 ONLINE 选项，表示该操作是联机操作，建立索引时不锁定数据，一个用户正在使用联机索引操作重新生成聚集索引时，该用户和其他用户可以继续更新和查询基础数据。联机索引操作期间会使用下列结构。

- 源和预先存在的索引：源是指原始表或聚集索引数据。预先存在的索引是指与源结构相关联的任何非聚集索引。
- 目标索引：目标是指正在创建或重新生成的新索引（或堆）或一组新索引。在索引操作期间，用户对源的插入、更新和删除操作，是由 SQL Server 数据库引擎应用到目标的。
- 临时映射索引：用于创建、删除或重新生成聚集索引的联机索引操作，还需要用到临时映射索引。此临时索引由并发事务来确定当更新或删除了基础表中的行以后，要从正在生成的新索引中删除哪些记录。此非聚集索引是在创建新聚集索引（或堆）的步骤中创建的，不需要执行单独的排序操作。并发事务还会在它们的所有插入、更新和删除操作中维护临时映射索引。

在简单的联机索引操作期间，源和目标将经历 3 个阶段，即准备阶段、生成阶段和最后阶段。如图 19.8 所示为联机创建初始聚集索引的过程。源对象没有其他索引。图 19.8 中分别显示了每个阶段的源结构和目标结构活动，另外还显示了并发的用户选择、插入、更新和删除操作。准备、生成和最后阶段均与每个阶段使用的锁模式一起指明。

联机索引操作过程中，在准备阶段和最终阶段表是被锁定的，并发用户不可进行操作，但是由于这两个阶段相对生成阶段来说时间非常短，所以在大部分时间里并发用户是可以对表进行操作的。

例如要对学生表中的索引进行联机情况下的重建，对应的 SQL 脚本如代码 19.10 所示。

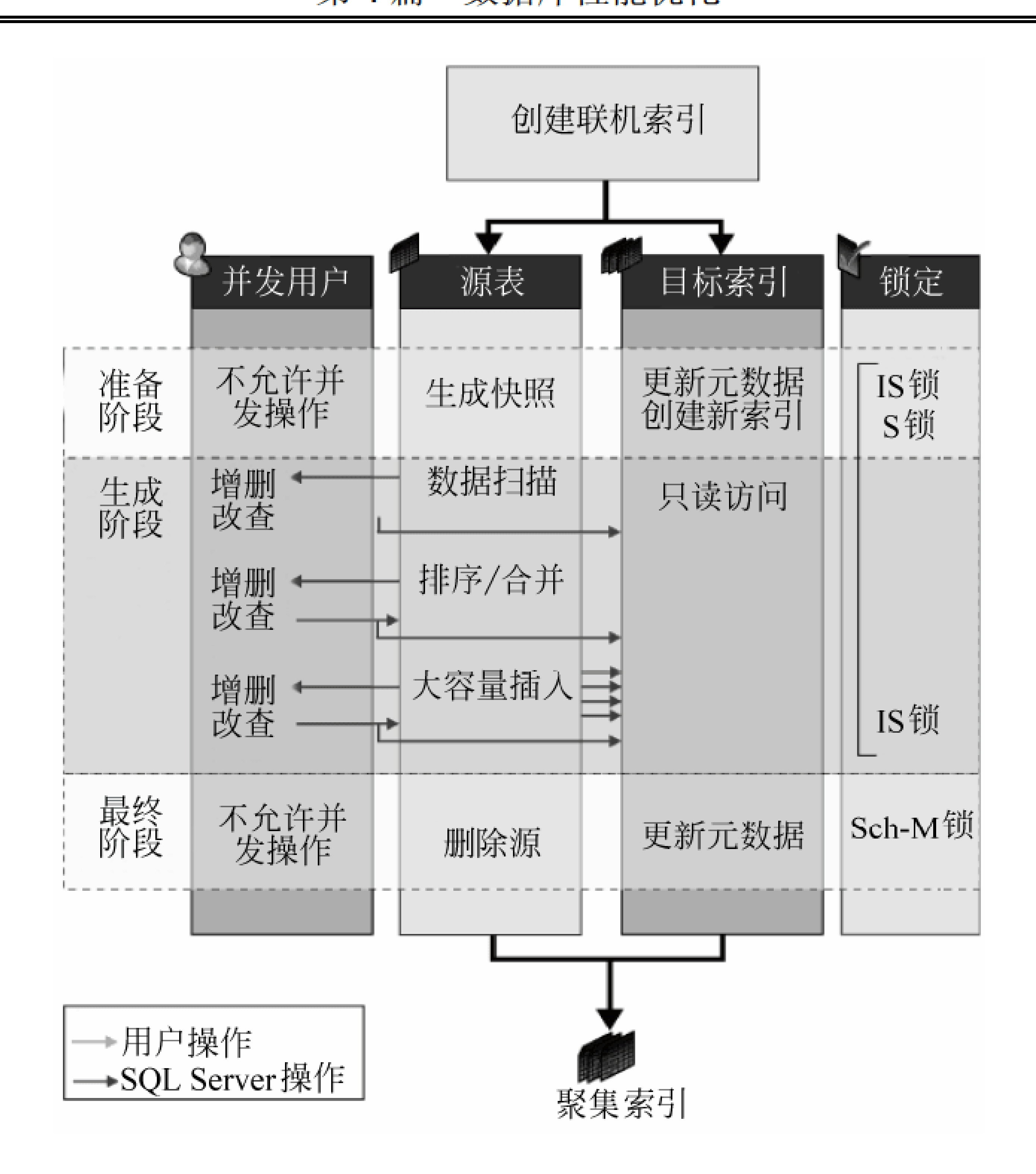


图 19.8　联机索引创建过程

代码 19.10　联机索引操作

```
ALTER INDEX ALL
ON dbo.Student
REBUILD WITH (ONLINE = ON)
```

可以联机创建包括全局临时表上的索引在内的索引，但下列索引例外：

- XML 索引。
- 对局部临时表的索引。
- 视图唯一的初始聚集索引。
- 已禁用的聚集索引。
- 聚集索引，前提是基础表包含 LOB 数据类型，如 image、ntext、text、varchar(max)、nvarchar(max)、varbinary(max)和 XML。
- 使用 LOB 数据类型列定义的非聚集索引。

注意：联机索引操作仅在 SQL Server 2005 企业版及更高版本中可用。

19.3.3　其他高级选项

ALLOW_ROW_LOCKS 是在 SQL Server 2005 中添加的一个新索引属性，用于确定访问索引数据时是否使用行锁。ALLOW_PAGE_LOCKS 也是 2005 版中增加的新属性，用于确定访问索引数据时是否使用页锁。

MAXDOP 为 SQL Server 2005 中新增的属性，为企业版所有，用于设置查询处理器最

多可以使用多少个处理器来执行单一索引描述。可以避免服务器内的资源同时进行同一项工作，而影响其他的操作。默认情况下会根据目前的系统工作负荷而定，所使用的处理器数量不确定。

在运行 SQL Server 企业版的多处理器计算机上，索引语句可能会像其他查询那样，使用多个处理器来执行与其关联的扫描、排序和索引操作。用于运行单个索引语句的处理器数是由配置选项 max degree of parallelism、当前工作负荷及索引统计信息决定的。max degree of parallelism 选项决定了执行并行计划时使用的最大处理器数。

查询优化器使用的处理器数量通常能够提供最佳的性能，但是有些操作（如创建、重新生成或删除很大的索引）占用大量资源，在索引操作期间会造成没有足够的资源供其他应用程序和数据库操作使用。出现此问题时，可以指定 MAXDOP 索引选项并限制用于索引操作的处理器数，手动配置用于运行索引语句的最大处理器数。

MAXDOP 索引选项只为指定此选项的查询覆盖 max degree of parallelism 配置选项。如表 19.2 列出了可为 max degree of parallelism 配置选项和 MAXDOP 索引选项指定的有效整数值。

表 19.2　MAXDOP索引选项可用的设置值

值	说　明
0	根据当前系统工作负荷，由系统自己选择使用实际可用的CPU数量。这是默认值，也是推荐设置
1	取消生成并行计划。只使用1个CPU，操作将以串行方式执行
2～64	将处理器的数量限制为指定的值。根据当前工作负荷，可能使用较少的处理器。如果指定的值大于可用的CPU数量，将使用实际可用的CPU数量

SORT_IN_TEMPDB 确定对创建索引期间生成的中间排序结果进行排序的位置。如果为 ON，排序结果存储在 tempdb 中；如果为 OFF，则排序结果存储在存储结果索引的文件组或分区方案中。

IGNORE_DUP_KEY 指定对唯一聚集索引或唯一非聚集索引的多行 INSERT 事务中，重复键值的错误响应。

19.4　数据文件分区

以往数据库文件只能是一个文件，一个数据库中的所有数据库对象和数据都存放在该文件中，数据文件过大会导致查找数据变慢。数据文件分区是 SQL Server 2005 新增加的一个特性，对数据文件进行分区有助于提供数据查找的性能。

19.4.1　分区概述

如果一个数据库实例中运行的数据较多，或者一个数据库中表的数据量太大，将数据库分区可以提高其性能并易于维护。对于大数据量的表，将表拆分为多个较小的表，如果访问的数据量较小，而且集中在某个分区中，则只访问部分数据的查询可以使运行速度更快，因为要扫描的数据变得更少。

分区并不一定要拆分表到多个文件上，如果使用 SQL Server 文件组指定放置表的磁

盘，将某个表放置在一个物理驱动器上，将相关表放置在另一个驱动器上，同样也可以提高查询性能，因为当运行涉及表间联接的查询时，多个磁盘头将同时读取数据。数据库的分区分为 3 种，下面分别进行介绍。

1．硬件分区

硬件分区将数据库设计为利用可用的硬件体系结构。例如，如果硬件条件是多 CPU 的服务器，那么可以同时运行许多查询。

在数据存储上，使用 RAID（独立磁盘冗余阵列）设备可以使得数据在多个磁盘驱动器中条带化，在读取数据时可以使用更多的读写磁头进行同时读写，因此可以更快地访问数据。如果一个服务器上有多个磁盘驱动器，那么将某个表与和它相关的表分开存储在不同的驱动器上，可以显著提高联接这些表查询的性能。

2．水平分区

水平分区将表中的行分为多个表，但是表的列并没有变。水平分区后表中的行更少。例如，可以将一个包含几千万甚至十亿行的业务表按照业务发生日期进行水平分区，将每一年的数据放入一个表，如果要查询某段时间的数据，只需要查询相关的表即可。

具体如何将表进行水平分区，取决于如何分析数据。对表进行分区应该让查询引用的表尽可能少，否则查询时需要使用过多的 UNION 查询来逻辑合并表，这会影响查询性能。

最常用的水平分区方法是根据时期和分类对数据进行的。例如，一个表可能包含最近 5 年的数据，但是只定期访问本年度的数据。再如一个表中包含了 10 种类型的产品订单，可以对每种类型分一个区，一般查询都是针对某一种类型进行，也就只进行一个表的查询。

3．垂直分区

垂直分区将一个表分为多个表，每个表中的数据行数相同，但包含较少的列。垂直分区包括两种类型，即规范化和行拆分：

- 规范化就是指数据库设计要符合三范式，它删除表中的多余列，并将这些列放置在通过主键和外键关系链接到主表的辅助表中。
- 行拆分将原始表垂直分成多个只包含较少列的表。拆分表内的每个逻辑行都与其他表内由唯一值列标识的相同逻辑行相匹配。

前面已经讲到数据库中的最小单位是页，一个页只有 8KB 大小，如果列少，一行数据占用的空间小，则一个页中可以存放更多行的数据，所以垂直分区的目的是使得查询需要扫描的页减少，这将提高查询性能。例如，某个表包含 7 列，通常只引用该表的前 4 列，那么将该表的后 3 列拆分到一个单独的表中将有利于提高性能。

注意： 尽量不要将查询使用的列分布到多个分区中，因为分析多个分区中的数据时需要联接表的查询。如果数据量过大，垂直分区还可能会影响性能。

19.4.2　文件和文件组

每个 SQL Server 数据库至少具有两个操作系统文件：一个数据文件和一个日志文件。

数据文件包含数据和对象，例如表、索引、存储过程和视图。日志文件包含恢复数据库中的所有事务所需的信息。为了便于分配和管理，可以将数据文件集合起来，放到文件组中。SQL Server 数据库具有 3 种类型的文件，如表 19.3 所示。

表 19.3　数据库文件类型

文　件	说　明
主要文件	主要数据文件包含数据库的启动信息，并指向数据库中的其他文件。用户数据和对象可存储在此文件中，也可以存储在次要数据文件中。每个数据库有一个主要数据文件。主要数据文件的建议文件扩展名是.mdf
次要文件	次要数据文件是可选的，由用户定义并存储用户数据。一般将每个文件放在不同的磁盘驱动器上，次要文件可用于将数据分散到多个磁盘上。另外，如果数据库超过了单个Windows文件的最大大小，可以使用次要数据文件，这样数据库就能继续增长。次要数据文件的建议文件扩展名是.ndf
日志文件	事务日志文件保存用于恢复数据库的日志信息。每个数据库必须至少有一个日志文件。事务日志文件的建议文件扩展名是.ldf

每个数据库有一个主要文件组。此文件组包含主要数据文件和未放入其他文件组的所有次要文件。可以创建用户定义的文件组，用于将数据文件集合起来，以便于对这些文件进行管理、数据分配和放置。

例如要创建一个数据库，该数据库中除了具有默认的文件组和其他的主要文件外，还有文件组 FG1，在该文件组中有主要文件 File1、File2、File3，则对应的创建该数据库的 SQL 脚本如代码 19.11 所示。

代码 19.11　创建文件组和其中的文件

```
CREATE DATABASE [PartitionTest] ON  PRIMARY
( NAME = N'PrimaryFile', FILENAME = N'C:\DATA\PrimaryFile.mdf' ),
 FILEGROUP [FG1]
( NAME = N'File1', FILENAME = N'C:\DATA\File1.ndf' ),
( NAME = N'File2', FILENAME = N'C:\DATA\File2.ndf' ),
( NAME = N'File3', FILENAME = N'C:\DATA\File3.ndf' )
 LOG ON
( NAME = N'PartitionTest_log', FILENAME = N'C:\DATA\PartitionTest_log.ldf')
```

在创建了文件组和文件后，便可将表创建在指定的文件组中。例如要创建一个表在 FG1 文件组中，对应的 SQL 脚本如代码 19.12 所示。

代码 19.12　创建指定文件组的表

```
CREATE TABLE tb1
(
    ID INT IDENTITY PRIMARY KEY ,
    NAME NVARCHAR(10) NOT NULL
)
ON FG1 --指定文件组
```

在使用文件和文件组时可以参照以下建议：

- 对于一般的数据库，在只有单个数据文件和单个事务日志文件的情况下性能良好。
- 如果使用多个文件，最好为附加文件创建第二个文件组，并将其设置为默认文件组。这样，主文件将只包含系统表和对象。

- 若要使性能最大化，应该尽可能多地在不同的可用本地物理磁盘上创建文件或文件组，这样将争夺空间最激烈的对象置于不同的文件组中。
- 使用文件组将对象放置在特定的物理磁盘上。
- 将在同一联接查询中使用的不同表置于不同的文件组中。由于采用并行磁盘 I/O 对联接数据进行搜索，所以性能得以改善。
- 将最常访问的表和属于这些表的非聚集索引置于不同的文件组中。如果文件位于不同的物理磁盘上，并且采用并行 I/O，性能将得以改善。
- 最好将事务日志文件与已有其他文件和文件组的物理磁盘分开。

对于现有的数据库，也可以通过 ALTER DATABASE 来添加文件和文件组。例如在前面创建的数据库 PartitionTest 中，添加文件组 FG2 和该文件组中的 3 个文件，对应的 SQL 脚本如代码 19.13 所示。

代码 19.13　修改数据库添加文件组和文件

```
USE [master]
GO
ALTER DATABASE [PartitionTest] ADD FILEGROUP [FG2]
GO
ALTER DATABASE [PartitionTest]
ADD FILE ( NAME = N'File21', FILENAME = N'C:\DATA\File21.ndf') TO FILEGROUP
[FG2]
GO
ALTER DATABASE [PartitionTest]
ADD FILE ( NAME = N'File22', FILENAME = N'C:\DATA\File22.ndf') TO FILEGROUP
[FG2]
GO
ALTER DATABASE [PartitionTest]
ADD FILE ( NAME = N'File23', FILENAME = N'C:\DATA\File23.ndf') TO FILEGROUP
[FG2]
GO
```

19.4.3　分区函数

在对表或索引进行分区前，必须计划创建下列数据库对象：

- 分区函数，定义如何根据某些列（称为分区依据列）的值将表或索引的行映射到一组分区。
- 分区方案，将把分区函数指定的每个分区映射到文件组。

创建分区函数使用 CREATE PARTITION FUNCTION 语句，其语法如代码 19.14 所示。

代码 19.14　创建分区函数语法

```
CREATE PARTITION FUNCTION partition_function_name ( input_parameter_type )
AS RANGE [ LEFT | RIGHT ]
FOR VALUES ( [ boundary_value [ ,...n ] ] )
```

其中几个参数的含义是：

- partition_function_name 是分区函数的名称。分区函数名称在数据库内必须唯一，并且符合标识符的规则。

- input_parameter_type 是用于分区列的数据类型。当用作分区列时，除 text、ntext、image、xml、timestamp、varchar(max)、nvarchar(max)、varbinary(max)、别名数据类型或 CLR 用户定义数据类型外，所有数据类型均有效。实际列（也称为分区列）是在 CREATE TABLE 或 CREATE INDEX 语句中指定的。
- boundary_value 为使用 partition_function_name 的已分区表或索引的每个分区指定边界值。如果 boundary_value 为空，则分区函数使用 partition_function_name 将整个表或索引映射到单个分区。只能使用 CREATE TABLE 或 CREATE INDEX 语句中指定的一个分区列。
- LEFT|RIGHT 指定当间隔值由数据库引擎按升序从左到右排序时，boundary_value[,...n]属于每个边界值间隔的哪一侧（左侧还是右侧）。如果未指定，则默认值为 LEFT。

例如要创建一个分区函数，该函数以 int 数据类型来分区，以 10、100、1000 作为分界点，对应的 SQL 脚本如代码 19.15 所示。

代码 19.15　创建 int 数据类型的分区函数

```
USE PartitionTest
GO
CREATE PARTITION FUNCTION IntRangePF(int)
AS RANGE LEFT
FOR VALUES (10, 100, 1000)
```

该分区函数把整个 int 区间分为 4 个区：小于等于 10、大于 10 而且小于等于 100、大于 100 而且小于等于 1000 和大于 1000。

如果是按照时间进行分区，将 2007 年之前作为一个分区，2007 年到 2008 年作为一个分区，2008 年之后又作为一个分区，则对应的分区函数如代码 19.16 所示。

代码 19.16　创建 datetime 数据类型的分区函数

```
USE PartitionTest
GO
CREATE PARTITION FUNCTION DateRangePF (DATETIME)   --以时间分区的分区函数
AS RANGE LEFT
FOR VALUES ('2007-1-1','2008-1-1')
```

修改分区函数使用 ALTER PARTITION FUNCTION 语句。修改分区函数的语法如代码 19.17 所示。

代码 19.17　修改分区函数语法

```
ALTER PARTITION FUNCTION partition function name()
{
   SPLIT RANGE ( boundary value )
  | MERGE RANGE ( boundary value )
}
```

在修改分区函数的 ALTER PARTITION FUNCTION 语句中，应主要注意以下几个参数。

- partition_function_name：要修改的分区函数的名称。

- SPLIT RANGE(boundary_value)：在分区函数中添加一个分区。boundary_value 确定新分区的范围，因此它必须不同于分区函数的现有边界范围。根据 boundary_value，数据库引擎将某个现有范围拆分为两个范围。在这两个范围中，新 boundary_value 所在的范围被视为是新分区。
- MERGE RANGE(boundary_value)：删除一个分区，并将该分区中存在的所有值都合并到剩余的某个分区中。

例如对于前面创建的 IntRangePF 分区函数，现在需要在其中减少分区，去掉 100 这个边界，对应的 SQL 脚本如代码 19.18 所示。

代码 19.18　修改分区函数

```
USE PartitionTest
GO
ALTER PARTITION FUNCTION IntRangePF()       --修改分区函数 IntRangePF
MERGE RANGE(100)                            --合并一个分区段
```

同样如要增加分区，则使用 SPLIT 关键字。ALTER PARTITION FUNCTION 只能用于将一个分区拆分为两个分区，或将两个分区合并为一个分区。若要在其他情况下对表进行分区，则可以使用以下方法。

- 使用所需的分区函数创建一个新的已分区表，然后使用 INSERT INTO…SELECT FROM 语句将旧表中的数据插入新表。
- 为堆创建分区聚集索引。
- 通过将 CREATE INDEX 语句与 DROP EXISTING = ON 子句一起使用，来删除并重新生成现有的已分区索引。
- 执行一系列 ALTER PARTITION FUNCTION 语句。

删除不需要的分区函数使用 DROP PARTITION FUNCTION 语句，例如要将 IntRangePF 分区函数删除，对应的脚本为：

```
DROP PARTITION FUNCTION IntRangePF
```

注意： 只有未使用的分区才能被删除，如果已被使用则会删除失败。

19.4.4　分区方案

在当前数据库中创建一个将已分区表或已分区索引的分区映射到文件组的方案。已分区表或已分区索引的分区的个数和域，在分区函数中确定。

分区方案必须在创建好分区函数后才能创建，创建分区方案使用 CREATE PARTITION SCHEME 语句，其语法如代码 19.19 所示。

代码 19.19　创建分区方案的语法

```
CREATE PARTITION SCHEME partition_scheme_name
AS PARTITION partition_function_name
[ ALL ] TO ( { file_group_name | [ PRIMARY ] } [ ,...n ] )
```

其中各个参数的含义如下所示。

- partition_scheme_name：分区方案的名称。
- partition_function_name：使用分区方案的分区函数的名称。分区函数所创建的分区将映射到在分区方案中指定的文件组。partition_function_name 必须已经存在于数据库中。单个分区不能同时包含 FILESTREAM 和非 FILESTREAM 文件组。
- ALL：指定所有分区都映射到在 file_group_name 中提供的文件组，或映射到主文件组（如果指定了[PRIMARY]。如果指定了 ALL，则只能指定一个 file_group_name。
- file_group_name|[PRIMARY][,...n]：指定用来持有由 partition_function_name 指定分区的文件组名称。file_group_name 必须已经存在于数据库中。

前面创建的数据库 PartitionTest 中现在有 3 个文件组，而分区函数 DateRangePF 正好也将数据分为 3 个区间，所以可在 PartitionTest 数据库中创建分区方案，将每个区间分别对应一个文件组，具体 SQL 脚本如代码 19.20 所示。

代码 19.20　创建分区方案

```
USE PartitionTest
GO
CREATE PARTITION SCHEME DateRangePS          --创建分区方案
AS PARTITION DateRangePF                     --分区函数和对应的文件组
TO ([PRIMARY],FG1, FG2);
```

一张表最多可以有 1000 个分区。不同的分区可以指定同一个文件组，甚至将所有的分区指定为同一个文件组只需要使用 ALL 关键字即可。

注意：除了使用 ALL 关键字指定所有分区对应一个文件组外，分别为分区指定文件组时两者数量必须相同，即使其中多个分区公用一个文件组，也要分别指明对应的文件组。

19.4.5　分区表

在创建了分区函数和分区方案后，便可将表创建在分区方案上。创建分区表的方法与创建普通表的方法相同，只是在指定文件组时指定分区方案和用于分区方案的列即可。例如要创建一个订单表 Orders，其中有下订单的时间列 CreateTime，通过该列在分区方案 DateRangePS 上创建订单表的 SQL 脚本如代码 19.21 所示。

代码 19.21　创建分区表

```
CREATE TABLE Orders
(
    ID INT IDENTITY,
    CreateTime DATETIME,
    Code VARCHAR(10),
    TotalMoney MONEY
)
ON DateRangePS(CreateTime)  --分区表，以 CreateTime 字段进行分区
```

创建好分区表后，接下来需要向该表中插入一些数据，然后查看这些数据是不是被插入到指定的分区中。要查看数据所在的分区，需要使用$PARTITION 函数。该函数的语法格式为：

```
[database_name.]$PARTITION.partition_function_name(expression)
```

其中的参数含义如下。

- database_name：包含分区函数的数据库名称。
- partition_function_name：对其应用一组分区列值的任何现有分区函数的名称。
- expression：其数据类型必须匹配或可隐式转换为其对应分区列数据类型的表达式。expression 也可以是当前参与 partition_function_name 的分区列的名称。

例如，向 Orders 表中插入一定数据，并查询分区的 SQL 脚本如代码 19.22 所示。

代码 19.22　为分区表插入数据并查询分区

```
INSERT INTO Orders
VALUES ('2008-8-1','2008000001',100 )
INSERT INTO Orders
VALUES ('2007-3-1','2007000001',1100 )
INSERT INTO Orders
VALUES ('2006-8-11','2006000001',1010 )
INSERT INTO Orders
VALUES ('2007-12-31','2007000011',2100 )
--接下来查询结果
SELECT Code, $PARTITION.[DateRangePF](CreateTime)
FROM Orders
ORDER BY Code
```

系统返回结果如下：

```
Code      (无列名)
2006000001  1
2007000001  2
2007000011  2
2008000001  3
```

19.4.6　分区索引

尽管可以从已分区索引的基表中单独实现已分区索引，但通常的做法是先设计一个已分区表，然后为该表创建索引。执行此操作时，SQL Server 将使用与该表相同的分区方案和分区依据列自动对索引进行分区。因此，索引的分区方式实质上与表的分区方式相同。

如果在创建时指定了不同的分区方案或单独的文件组来存储索引，则 SQL Server 不会将索引与表对齐。如果预计将通过使用更多分区来扩展索引，或者将会涉及频繁的分区切换，那么将索引与已分区表对齐非常重要。在下列情况下，独立于基表而单独设计已分区索引很有用：

- 基表未分区。
- 索引键是唯一的，不包含表的分区依据列。
- 希望基表与使用不同联接列的多个表一起参与组合联接。

例如在 Orders 表中创建独立于基表的分区索引，以 Code 列为唯一聚集索引，对应的

SQL 脚本如代码 19.23 所示。

代码 19.23　创建独立于基本的分区索引

```
ALTER DATABASE [PartitionTest] ADD FILEGROUP [FGIndex] --创建文件组 FGIndex
GO
ALTER DATABASE [PartitionTest]                          --添加文件到文件组
ADD FILE ( NAME = N'FileIndex', FILENAME = N'C:\DATA\File Index.ndf')
TO FILEGROUP [FGIndex]
GO
CREATE UNIQUE NONCLUSTERED INDEX CIX_Orders_Code        --创建非聚集索引
ON Orders(Code)
ON FGIndex    --指定分区索引在该 FGIndex 文件组中
```

索引要与其基表对齐，并不需要与基表参与相同的命名分区函数。但是，索引和基表的分区函数在本质上必须相同，即：

- 分区函数的参数具有相同的数据类型。
- 分区函数定义了相同数目的分区。
- 分区函数为分区定义了相同的边界值。

创建与基表对齐的分区索引时，需要注意：

- 对唯一索引（聚集或非聚集）进行分区时，必须从唯一索引键使用的分区依据列中选择分区依据列。
- 对聚集索引进行分区时，聚集键必须包含分区依据列。对非唯一的聚集索引进行分区时，如果未在聚集键中明确指定分区依据列，默认情况下 SQL Server 将在聚集索引键列表中添加分区依据列。如果聚集索引是唯一的，则必须明确指定聚集索引键包含分区依据列。
- 对唯一的非聚集索引进行分区时，索引键必须包含分区依据列。对非唯一的非聚集索引进行分区时，默认情况下 SQL Server 将分区依据列添加为索引的非键（包含性）列，以确保索引与基表对齐。如果索引中已经存在分区依据列，SQL Server 将不会向索引中添加分区依据列。

例如为 Orders 表创建与基表对齐的分区索引，仍然使用 DateRangePS 作为分区方案，对应的 SQL 脚本如代码 19.24 所示。

代码 19.24　创建与基表对齐的分区索引

```
CREATE UNIQUE NONCLUSTERED INDEX CIX_Orders_ID ON dbo.Orders
(
    ID ASC,
    CreateTime ASC
)
ON DateRangePS(CreateTime)  --使用分区函数创建分区索引
```

19.5　全 文 搜 索

SQL Server 2012 包含针对 SQL Server 表中基于纯字符的数据发出全文查询的功能。

全文查询可以包括词和短语，或者词或短语的多种形式。

19.5.1　全文搜索概述

对于 SQL Server 中字符串的查找，一般使用 LIKE 操作符，但是对于新闻正文、论坛贴、博客文章等使用 NVARCHR(MAX)或者 NTEXT 类型的数据进行 LIKE 查找，如果数据量较大时，可能需要花费几分钟乃至几十分钟。而使用 SQL Server 提供的全文搜索，使用 LIKE 需要几分钟才能完成的任务，只需要几秒钟或更短的时间就能完成。

通过使用全文搜索，可以快速且灵活地为 SQL Server 数据库中存储的文本数据的基于关键字的查询创建索引。与仅适用于字符模式的 LIKE 不同，全文查询将根据特定语言的规则对词和短语进行操作，从而针对此数据执行语言搜索。

注意：从 SQL Server 2008 版本开始，全文引擎位于 SQL Server 进程中，而不是像 SQL Server 代理一样位于单独的服务中。

全文索引的结构由以下 Microsoft Full-Text Engine for SQL Server（MSFTESQL）和 Microsoft Full-Text Engine Filter Daemon（MSFTEFD）组成。MSFTEFD 包含下列组件。

- 协议处理程序：此组件从内存中取出数据，以进行进一步的处理，并访问指定数据库的用户表中的数据。
- 筛选器：筛选器从文档中提取文本化信息流，并舍弃所有非文本化信息和格式信息。
- 断字符和词干分析器：这些组件对所有全文索引数据执行语言分析。“断字符”用于确定在进行全文索引的行或文档中的单词边界位于文本流中的什么位置。“词干分析器”提取给定单词的根形式。

SQL Server 中的全文搜索结构如图 19.9 所示。

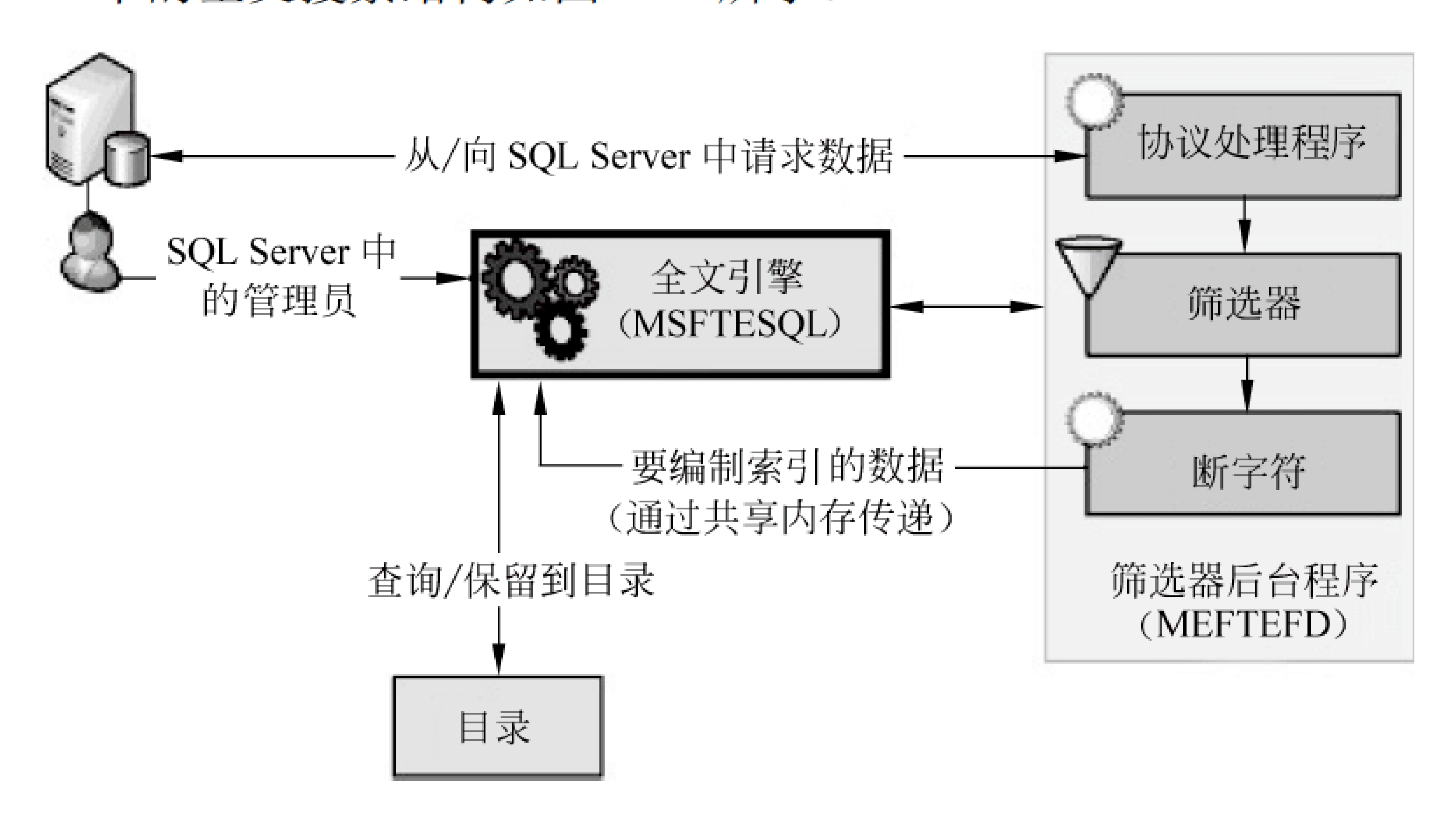


图 19.9　全文搜索的结构

全文搜索的索引组件独立于 SQL Server 数据库引擎，以独立的进程存在。索引组件负责对全文索引进行初始填充，并在以后修改全文索引表中的数据后对该索引进行更新。全

文填充开始后，全文引擎会将用户指定的列中的数据大批地存入到内存中，并通知筛选器后台程序宿主。宿主负责对数据进行筛选和断字，并将转换的数据转换为倒排词列表。接下来，全文搜索从词列表中提取转换的数据，对其进行处理以删除非索引字，然后将某一批次的词列表永久保存到一个或多个倒排索引中。

全文搜索不仅仅可以用于字符串，还可用于二进制数据的搜索。若要对存储在 varbinary(max)或 image 列中的数据编制索引，只要实现筛选器 IFilter 接口，将基于为该数据指定的文件格式（例如 Word）来提取文本。在某些情况下，筛选器组件要求将 varbinary(max)或 image 数据写入 filterdata 文件夹中，而不是将其存入内存。在处理过程中，通过断字符将收集到的文本数据分隔成各个单独的标记或关键字，还可能会进行其他处理以删除非索引字，并在标记存储到全文索引或索引片段之前对其进行规范化。

在全文索引填充完成后，将触发最终的合并过程，系统将索引片段合并为一个主全文索引。由于只需要查询主索引而不需要查询大量索引片段，因此将提高查询性能，并且可以使用更好的计分统计信息得出相关性排名。

注意： 可以对包含 char、varchar 和 nvarchar 数据的列创建全文索引，也可以对包含格式化二进制数据（如存储在 varbinary(max)或 image 列中的 Microsoft Word 文档）的列创建全文索引。不能使用 LIKE 谓词来查询格式化的二进制数据。

19.5.2 全文目录

为某个表设置全文索引功能需要执行以下两个步骤。

（1）创建全文目录来存储全文索引。

（2）创建全文索引。

对于 SQL Server 数据库，全文目录是一个表示一组全文索引的逻辑概念。全文目录是虚拟对象，并不属于任何文件组。创建全文目录使用 CREATE FULLTEXT CATALOG 语句，其语法如代码 19.25 所示。

代码 19.25　创建全文目录的语法

```
CREATE FULLTEXT CATALOG catalog_name
     [ON FILEGROUP filegroup ]
     [WITH ACCENT_SENSITIVITY = {ON|OFF}]
     [AS DEFAULT]
     [AUTHORIZATION owner_name ]
```

其中几个参数的含义如下。

- catalog_name：新目录的名称。
- ON FILEGROUP 'filegroup'：包含新目录的 SQL Server 文件组的名称。如果未指定文件组，则新目录将包含在用于所有全文目录的默认文件组中。默认全文文件组是数据库的主文件组。
- ACCENT_SENSITIVITY={ON|OFF}：指定该目录的全文索引是否区分重音。在更改此属性后，必须重新生成索引。默认情况下，将使用数据库排序规则中所指定

的区分重音设置。

- AS DEFAULT：指定该目录为默认目录。如果在未显式指定全文目录的情况下创建全文索引，使用默认目录；如果现有全文目录已标记为 AS DEFAULT，则将新目录设置为 AS DEFAULT，将使该目录成为默认全文目录。

例如现在有 TestDB1 数据库，该数据库中使用的是默认的文件组，现在要为该数据库创建全文目录，对应的 SQL 脚本如代码 19.26 所示。

代码 19.26　创建全文目录

```
USE TestDB1
GO
ALTER DATABASE [TestDB1] ADD FILEGROUP [FTGroup]
GO
ALTER DATABASE [TestDB1]
ADD FILE ( NAME = N'FtsFile', FILENAME = N'C:\DATA\FtsFile.ndf') TO FILEGROUP
FTGroup
GO
CREATE FULLTEXT CATALOG FT Cat --创建全文目录
ON FILEGROUP FTGroup            --指定文件组
AS DEFAULT
```

与一般的索引类似，全文目录在创建后可以进行重建、重组等修改操作，修改全文目录使用 ALTER FULLTEXT CATALOG 命令，其语法如代码 19.27 所示。

代码 19.27　修改全文目录

```
ALTER FULLTEXT CATALOG catalog name
{ REBUILD [ WITH ACCENT SENSITIVITY = { ON | OFF } ]
| REORGANIZE
| AS DEFAULT
}
```

例如要重建全文目录 FT_Cat，对应的脚本为：

```
ALTER FULLTEXT CATALOG FT_Cat REBUILD
```

而删除全文目录，使用 DROP FULLTEXT CATALOG catalog_name 格式的语句即可。

在 SSMS 中，也可以通过可视化的操作来创建全文目录。在 SSMS 的对象资源管理器中展开对应数据库节点下的“存储”节点，选择其下的“全文目录”节点，在弹出的快捷菜单中选择“新建全文目录”选项，打开“新建全文目录”对话框，如图 19.10 所示。

注意：在可视化环境下无法设置全文目录对应的文件组。

19.5.3　全文索引

全文索引不同于普通的 SQL Server 索引，它是一种特殊类型的基于标记的功能性索引，是由 SQL Server 全文引擎生成和维护的。生成全文索引的过程也不同于生成其他类型的索引。

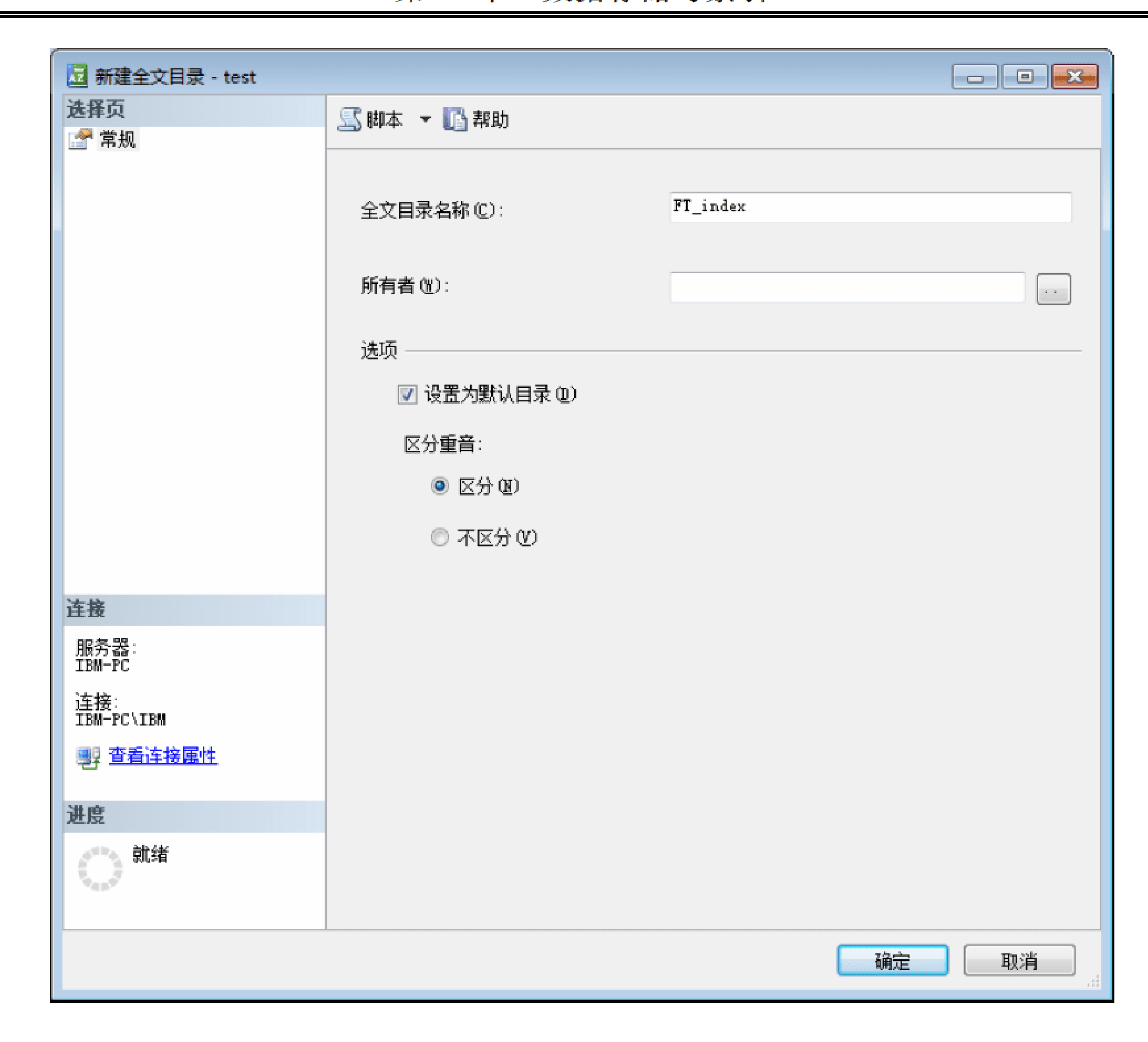


图 19.10　“新建全文目录”对话框

全文引擎并非基于特定行中存储的值来构造 B 树结构，而是基于要编制索引的文本中的各个标记来生成倒排、堆积且压缩的索引结构。

可以对字符串类型包含 char、varchar 和 nvarchar 数据的列生成全文索引，也可以为二进制类型 varbinary(max)或 image 列生成全文索引。

创建和维护全文索引的过程称为“索引填充”。SQL Server 支持下列全文索引填充：

- 完全填充，在全文目录的完全填充过程中，会对该目录涉及的所有表和索引视图中的所有行创建索引项。一般发生在首次填充全文目录或全文索引时。随后可以使用更改跟踪填充或增量填充来维护这些索引。
- 基于更改跟踪的填充，SQL Server 会记录设置了全文索引的表或索引视图中修改过的行。这些更改会被传播到全文索引。
- 基于时间戳的增量式填充，增量填充在全文索引中更新上次填充的当时或之后添加、删除或修改的行。增量填充要求索引表必须具有 timestamp 数据类型的列。如果 timestamp 列不存在，则无法执行增量填充。

注意：对不含 timestamp 列的表，请求增量填充会导致完全填充操作。

每个表或索引视图只允许有一个全文索引。该索引最多可包含 1024 列。创建全文索引使用 CREATE FULLTEXT INDEX 命令，其语法如代码 19.28 所示。

代码 19.28　创建全文索引语法

```
CREATE FULLTEXT INDEX ON table_name
     [ ( { column_name
```

```
              [ TYPE COLUMN type column name ]
              [ LANGUAGE language term ]
         } [ ,...n]
             ) ]
    KEY INDEX index name
        [ ON fulltext catalog name ]
        [ WITH [ ( ] <with option> [ ,...n] [ ) ] ]
<with option>::=
  {
   CHANGE TRACKING [ = ] { MANUAL | AUTO | OFF [, NO POPULATION ] }
  | STOPLIST [ = ] { OFF | SYSTEM | stoplist name }
  }
```

其中的参数含义如下。

- table_name：包含全文索引中的一列或多列的表或索引视图的名称。
- column_name：全文索引中包含的列的名称。只能对类型为 char、varchar、nchar、nvarchar、text、ntext、image、xml 和 varbinary 的列进行全文索引。
- TYPE COLUMN type_column_name：仅当 column_name 中的一个或多个列属于类型 varbinary(max)或 image 时，才指定 type_column_name；否则 SQL Server 将返回一个错误。
- LANGUAGE language_term：存储在 column_name 中的数据的语言。
- KEY INDEX index_name：指定 table_name 的唯一键索引的名称。
- fulltext_catalog_name：指定用于全文索引的全文目录。
- CHANGE_TRACKING[=]{MANUAL|AUTO|OFF[,NO POPULATION]}：指定 SQL Server 是否维护对索引数据的所有更改的列表。更改跟踪不会记录通过 WRITETEXT 和 UPDATETEXT 进行的数据更改。MANUAL 指定是使用 SQL Server 代理按计划传播更改跟踪日志，还是由用户手动进行传播。AUTO 指定在关联的表中修改了数据时，SQL Server 自动更新全文索引。默认值为 AUTO。OFF[,NO POPULATION]指定 SQL Server 不保留对索引数据更改的列表。

例如创建一个表 Fts1，该表中有主键列 ID 和 NVARCHAR(max)类型的 TSQL 列，现在要在该表上创建全文索引，对应的 SQL 脚本如代码 19.29 所示。

代码 19.29　创建全文索引

```
CREATE TABLE Fts1   --创建测试表
(
    ID INT IDENTITY,
    TSQL NVARCHAR(max) NOT NULL,
    CONSTRAINT PK_Fts1ID PRIMARY KEY CLUSTERED (ID)
)
GO
CREATE FULLTEXT INDEX ON Fts1([TSQL] LANGUAGE 'Simplified Chinese')
--创建全文索引
KEY INDEX PK_Fts1ID
ON FT_Cat
WITH CHANGE_TRACKING=AUTO
```

创建好全文索引后就需要在表中插入数据，然后将数据填充到全文索引中。填充全文索引需要使用 ALTER FULLTEXT INDEX 语句，其语法格式如代码 19.30 所示。

代码 19.30　修改全文索引语法

```
ALTER FULLTEXT INDEX ON table_name
   { ENABLE
   | DISABLE
   | SET CHANGE TRACKING { MANUAL | AUTO | OFF }
   | ADD ( column name
     [ TYPE COLUMN type_column_name ]
     [ LANGUAGE language term ] [,...n] )
     [ WITH NO POPULATION ]
   | DROP ( column name [,...n] )
     [WITH NO POPULATION ]
   | START { FULL | INCREMENTAL | UPDATE } POPULATION
   | {STOP | PAUSE | RESUME } POPULATION
   | SET STOPLIST { OFF| SYSTEM | stoplist name }
     [WITH NO POPULATION]
   }
```

其中比较重要的参数含义如下。

- ENABLE|DISABLE：通知 SQL Server 是否收集 table_name 的全文索引数据。ENABLE 为激活全文索引；DISABLE 为关闭全文索引。
- ADD|DROP column_name：指定要添加到全文索引或从全文索引中删除的列。
- START{FULL|INCREMENTAL|UPDATE}POPULATION：通知 SQL Server 开始填充 table_name 的全文索引。如果全文索引填充已在执行，SQL Server 将返回一个警告，并且不会启动新的填充。
- {STOP|PAUSE|RESUME}POPULATION：停止或暂停正在进行的任何填充，或者停止或恢复任何暂停的填充。

例如对该全文索引进行全部填充，对应的脚本如代码 19.31 所示。

代码 19.31　为全文索引启用完全填充

```
ALTER FULLTEXT INDEX ON Fts1
START FULL POPULATION   --完全填充
```

技巧： SSMS 中提供了创建全文搜索向导，右击要创建的表，在右键菜单中的“全文索引”选项下提供了创建全文索引、填充全文索引、修改全文索引等选项。

19.5.4　使用全文搜索

CONTAINSTABLE()和 FREETEXTTABLE()函数用来指定全文查询，以返回每行的相关性排名。这两个函数与全文谓词 CONTAINS 和 FREETEXT 非常相似，但是用法不同。

虽然全文谓词和全文函数都用于全文查询，且二者用来指定全文搜索条件的语法是一样的，但是它们在使用方法上仍有显著差别。下面列出了一些重要的相似处和不同处。

- CONTAINS 和 FREETEXT 都返回 TRUE 或 FALSE 值，并且都在 SELECT 语句的 WHERE 或 HAVING 子句中指定。
- CONTAINSTABLE( )和 FREETEXTTABLE( )都返回包含 0 行、一行或多行的表，因此它们必须始终在 FROM 子句中指定。

- ❑ CONTAINS 和 FREETEXT 只能用于指定选择条件，Microsoft SQL Server 将使用该条件来确定结果集的成员身份。
- ❑ CONTAINSTABLE()和 FREETEXTTABLE()也用来指定选择条件。返回的表中有一个名为 KEY 的列，其中包含全文键值。每个全文注册表必定会有某一列的值是唯一的。在 CONTAINSTABLE()或 FREETEXTTABLE()返回的全文注册表中，KEY 列中的值是与全文搜索条件中所指定的选择条件匹配的行的唯一值。

此外，CONTAINSTABLE()和 FREETEXTTABLE()生成的表中还有一个名为 RANK 的列，其中包含 0～1000 的值。值越小，相关性越低。根据返回的行与选择条件的匹配程度，使用这些值对行进行排名。

CONTAINS 和 FREETEXT 谓词可以与其他任一 T-SQL 谓词（如 LIKE 和 BETWEEN）相结合，它们还可用于子查询中。例如要在 Fts1 表中找到含有“化妆品”3 个字的行，对应的 SQL 脚本如代码 19.32 所示。

代码 19.32　使用 CONTAINS( )和 FREETEXT( )函数进行全文搜索

```
SELECT *
FROM Fts1
WHERE CONTAINS([TSQL],N'化妆品') --全文搜索函数
GO
SELECT *
FROM Fts1
WHERE FREETEXT ([TSQL],N'化妆品') --全文搜索函数
```

说明：使用 FREETEXT 的全文查询没有使用 CONTAINS 的全文查询精度高。SQL Server 全文搜索引擎识别重要的字词和短语。保留关键字或通配符字符都不具有特殊含义，而这些关键字和通配符作为 CONTAINS 的参数时则通常有了含义。

CONTAINSTABLE()函数和 FREETEXTTABLE()函数的使用相对复杂，而且返回的是索引的 KEY 和排名，所以还要和源表进行连接才能获得数据。同样是查询 Fts1 表中的与化妆品有关的内容，使用这两个函数的 SQL 查询如代码 19.33 所示。

代码 19.33　使用 CONTAINSTABLE( )和 FREETEXTTABLE( )函数全文检索

```
SELECT *
FROM Fts1 INNER JOIN CONTAINSTABLE(Fts1,[TSQL],N'化妆品') ct   --全文搜索
ON Fts1.ID=ct.[KEY]
ORDER BY ct.[Rank] DESC
GO
SELECT *
FROM Fts1 INNER JOIN FREETEXTTABLE(Fts1,[TSQL],N'化妆品') ft   --全文搜索
ON Fts1.ID=ft.[KEY]
ORDER BY ft.[Rank] DESC
```

19.6　使用 FILESTREAM 存储文件

FILESTREAM 是在 SQL Server 2008 版本中新增的功能，允许以独立文件的形式存放大对象数据，而不是像以往一样将所有数据都保存到数据文件中。本节将主要讲解

FILESTREAM 的配置和使用。

19.6.1　FILESTREAM 概述

以往在对业务系统的文件进行管理时有两种方法，一种是将文件保存到服务器文件系统中，数据库中只保存了该文件的路径，在使用该文件时应用程序连接到服务器读取文件；另一种是将文件以 varbinary(max)或 image 数据类型保存到 SQL Server 中。而 SQL Server 2008 提供了 FILESTREAM，其结合了以上这两种方式的优点。

FILESTREAM 使 SQL Server 数据库引擎和 NTFS 文件系统成为了一个整体。Transact-SQL 语句可以插入、更新、查询、搜索和备份 FILESTREAM 数据。FILESTREAM 使用 NT 系统缓存来缓存文件数据。这有助于减少 FILESTREAM 数据可能对数据库引擎性能产生的任何影响。由于没有使用 SQL Server 缓冲池，因此该内存可用于查询处理。

在 SQL Server 中，BLOB 可以是将数据存储在表中的标准 varbinary(max)数据，也可以是将数据存储在文件系统中的 FILESTREAM varbinary(max)对象。数据的大小和应用情况决定应该使用数据库存储还是文件系统存储。如果满足以下条件，则应考虑使用 FILESTREAM：

- 所存储的对象平均大于 1MB。
- 快速读取访问很重要。
- 开发的是使用中间层作为应用程序逻辑的应用程序。

对于较小的对象，将 varbinary(max)BLOB 存储在数据库中，通常会提供更为优异的流性能。

FILESTREAM 存储以 varbinary(max)列的形式实现，在该列中数据以 BLOB 的形式存储在文件系统中。BLOB 的大小仅受文件系统容量大小的限制。文件大小为 2GB 的 varbinary(max)标准限制不适用于存储在文件系统中的 BLOB。

若要将指定列使用 FILESTREAM 存储在文件系统中，对 varbinary(max)列指定 FILESTREAM 属性。这样数据库引擎会将该列的所有数据存储在文件系统中，而不是数据库文件中。

FILESTREAM 数据必须存储在 FILESTREAM 文件组中。FILESTREAM 文件组是包含文件系统目录而非文件本身的专用文件组。这些文件系统目录称为“数据容器”。数据容器是数据库引擎存储与文件系统存储之间的接口。

使用 FILESTREAM 存储时，需要注意以下内容：

- 如果表包含 FILESTREAM 列，则每一行都必须具有唯一的行 ID。
- 不能嵌套 FILESTREAM 数据容器。
- 使用故障转移群集时，FILESTREAM 文件组必须位于共享磁盘资源上。
- FILESTREAM 文件组可位于压缩卷上。

19.6.2　创建 FILESTREAM

在开始使用 FILESTREAM 之前，必须在 SQL Server 数据库引擎实例中启用 FILESTREAM。具体启用数据库实例 FILESTREAM 的操作如下。

（1）在 SQL Server 配置管理器中打开 SQL Server 数据库引擎的属性对话框，切换到

FILESTREAM 选项卡，如图 19.11 所示。

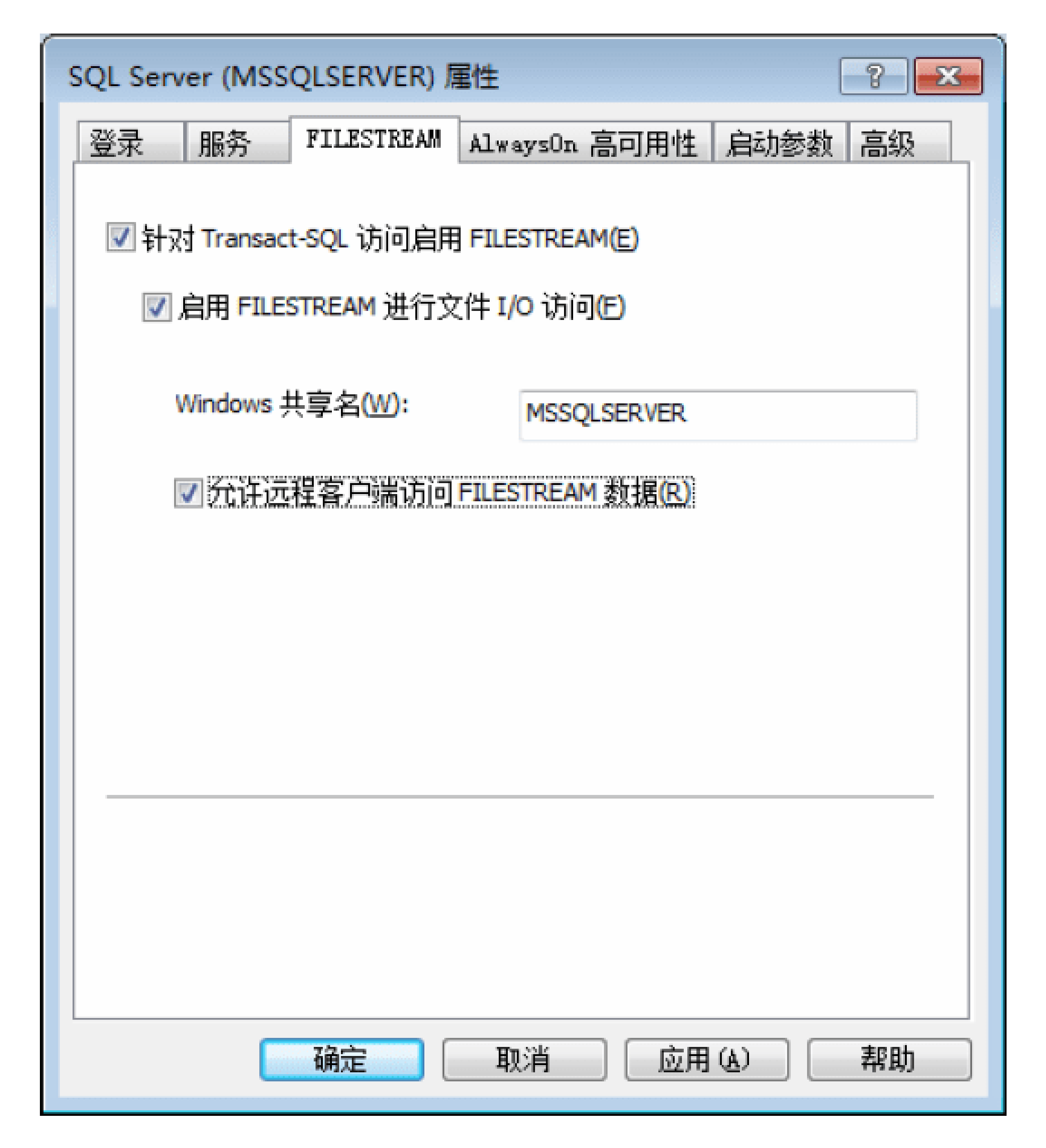


图 19.11　SQL Server 属性中的 FILESTREAM 配置

（2）选中“针对 Transact-SQL 访问启用 FILESTREAM”复选框，其他的选项是针对 Windows 进行读写的，可以都选中，然后单击“确定”按钮保存对 FILESTREAM 的设置。

（3）打开 SSMS 连接到数据库实例，右击数据库实例，在弹出的快捷菜单中选择“属性”选项，打开 SQL Server 实例的属性对话框。

（4）切换到“高级”选项页，在文件流访问级别下拉列表框中选择“已启用完全访问”选项，如图 19.12 所示。

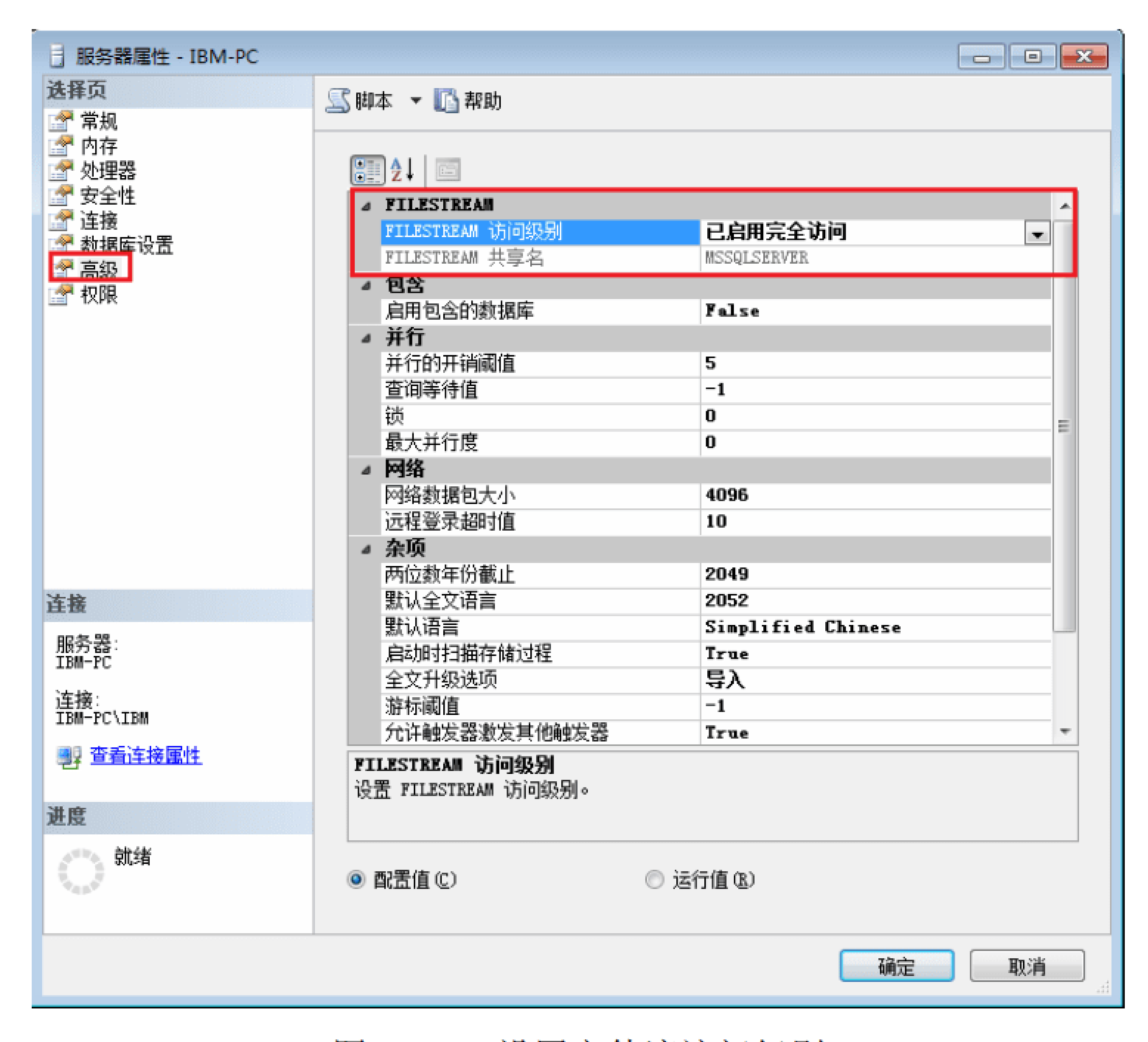


图 19.12　设置文件流访问级别

（5）单击“确定”按钮，然后重启数据库实例，FILESTREAM 在数据库实例中设置完成。

在启用了数据库实例的 FILESTREAM 后，接下来需要设置数据库的 FILESTREAM 及创建具有 FILESTREAM 数据列的表。

（6）对应新建的数据库，在创建数据库时创建 FILESTREAM 文件组；如果是现有数据库，则使用 ALTER DATABASE 添加 FILESTREAM 的文件组，例如对 TestDB1 数据库添加 FILESTREAM 的文件组，具体 SQL 脚本如代码 19.34 所示。

代码 19.34　为现有数据库添加 FILESTREAM 文件组

```
ALTER DATABASE [TestDB1]
ADD FILEGROUP [FileStreamGroup] CONTAINS FILESTREAM--添加 FILESTREAM 文件组
GO
ALTER DATABASE [TestDB1]
ADD FILE ( NAME = N'FileStream', FILENAME = N'C:\FileStream)
--添加 FILESTREAM 文件
TO FILEGROUP [FileStreamGroup]
GO
```

系统将自动创建 C:\FileStream 文件夹并在其中写入 filestream.hdr 文件，该文件是 FILESTREAM 容器的头文件，不能删除，一定要确保在运行该语句之前 C:\FileStream 并不存在。

（7）创建了 FILESTREAM 文件组后便可创建和修改表，指定某 varbinary(max)类型的列包含 FILESTREAM 数据。例如创建 Files 表，该表包含 FileID 和 FIleContent 列，具体脚本如代码 19.35 所示。

代码 19.35　创建具有 FILESTREAM 列的表

```
CREATE TABLE Files
(
    FileID UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL UNIQUE,
    ID INT UNIQUE,
    FileContent VARBINARY(MAX) FILESTREAM NULL --FILESTREAM 类型的二进制
)
```

19.6.3　管理与使用 FILESTREAM

在创建好 FILESTREAM 表后即可向其中添加、修改和读取数据。SQL Server 支持使用 T-SQL 和 WIN32 API 两种方式访问 FILESTREAM。

对于 T-SQL 访问 FILESTREAM 数据列来说，FILESTREAM 是完全透明的。也就是说，T-SQL 仍然使用一般的访问 varbinary(max)数据列的方式访问，并不会因为是 FILESTREAM 列而有所不同。

例如向 Files 表中插入数据、修改表数据和删除数据的 SQL 脚本，如代码 19.36 所示。

代码 19.36　操作 FILESTREAM 数据列

```
INSERT INTO Files   --插入测试数据
VALUES (newid (),1, CAST('TestFileStream1' as varbinary(max)));
GO
```

```
UPDATE Files          --更新测试数据
SET FileContent=CAST('TestFileStream2' as varbinary(max))
WHERE ID=1
GO
DELETE FROM Files     --删除测试数据
WHERE ID=1
```

无论是插入数据还是修改数据，SQL Server 都将在文件系统中创建新的文件来保存最新的修改文件内容，修改或删除数据后文件系统中的文件将保留，而不会被同时删除。

使用 FILESTREAM 存储二进制大型对象（BLOB）数据时，可使用 Win32 API 来处理文件。为了支持在 Win32 应用程序中处理 FILESTREAMBLOB 数据。所有 FILESTREAM 数据容器访问都是在 SQL Server 事务中执行的。可在同一事务中执行 T-SQL 语句以保持 SQL 数据和 FILESTREAM 数据之间的一致性。

19.7　小　　结

本章主要讲解了数据库中数据的存储和索引的基础知识。包括数据库页、索引、数据文件分区、全文搜索和文件流。

SQL Server 中数据存储的基本单位是页，一个页的大小为 8KB，数据都存放在页中。索引是采用 B 树结构存放用于检索数据的一种数据库对象。索引分为聚集索引和非聚集索引。SQL Server 中的数据文件可以进行分区，通过建立分区函数、分区方案，然后将表和索引进行分区，使得数据落在不同的文件组中，从而提高数据库的性能。SQL Server 提供了全文搜索功能，可以快速且灵活地为 SQL Server 数据库中存储的文本数据基于关键字的查询创建索引。最后介绍了 FILESTREAM 文件流，用于将大对象 varbinary(max)数据类型以独立文件的形式保存到文件系统中。

第 20 章　数 据 查 询

在数据库操作中，大部分都是进行数据查询，而由于数据查询涉及到的数据量大，数据关系复杂，所以该操作便成为了数据库性能中的一个关键。本章将主要讲解数据查询的相关知识，其中会涉及一些多表查询知识（如联接），这些知识点虽然复杂，但实际应用中必不可少。

20.1　执 行 计 划

T-SQL 语句在提交到数据库引擎后将编译成执行计划，然后根据执行计划的内容执行数据库操作，执行计划的好坏关系到最终数据操作效率的好坏，本节将主要讲解执行计划的相关知识。

20.1.1　执行计划缓存

SQL Server 将根据执行计划决定在一个 SQL 查询中该用到哪些索引和进行哪些操作。当一个 SQL 查询在运行时，系统将把提交的 T-SQL 语句编译成执行计划进行数据库操作。同时，SQL Server 中还有一个用于存储执行计划和数据缓冲区的内存池，执行计划将会被缓存在内存池中以便下次执行相同语句时不再进行编译。池内分配给执行计划或数据缓冲区的百分比，随系统状态动态波动。内存池中用于存储执行计划的部分称为过程缓存。

SQL Server 执行计划包含下列主要组件。

- 查询计划。执行计划的主体是一个只读的数据结构，可由任意数量的用户使用，这称为查询计划。查询计划中不存储用户上下文。
- 执行上下文。每个正在执行查询的用户都有一个包含其执行专用数据（如参数值）的数据结构。此数据结构称为执行上下文。执行上下文数据结构可以重新使用。如果用户执行查询而其中的一个结构未使用，将会用新用户的上下文重新初始化该结构。

在 SQL Server 中执行任何 SQL 语句时，关系引擎并不是马上对该语句进行分析编译，而是首先查看过程缓存中是否有用于同一 SQL 语句的现有执行计划。如果找到了对应的执行计划，则 SQL Server 将重新使用该执行计划，从而节省重新编译 SQL 语句的开销。如果没有找到现成的执行计划，SQL Server 将为查询生成新的执行计划，并且将其缓存起来。

SQL Server 内部有一个高效的算法，用于查找任何特定 SQL 语句的现有执行计划。在大多数系统中，执行这种查找扫描，然后重新使用现有计划所使用的资源，比重新编译每个 SQL 语句所消耗的资源要少。

执行计划生成后，将位于过程缓存中。仅当需要空间时，SQL Server 才会将缓存中旧的未使用的计划老化掉。每个执行计划和执行环境都有相关的成本因子用于表明编译结构所需的费用，另外还有个年龄字段表示执行计划的新旧程度。

执行计划每被重用一次，其年龄字段便按编译成本因子递增。例如，如果一个执行计划的成本因子为 8，并且被引用了 2 次，则其年龄变为 16。SQL Server 中还有一个惰性写入器进程，用于定期扫描过程缓存中的对象列表，每扫描一次对象的年龄减少 1。如果执行计划的年龄减小为 0，则该执行计划就可能从缓存中清除。如果满足下面 3 个条件，惰性写入器进程将释放对象。

- 内存管理器需要内存，而所有可用内存都正在使用。
- 对象的年龄字段是 0。
- 对象在当前没有被连接引用。

因为每次引用对象时其年龄字段都会增加，所以经常被引用的对象的年龄字段不会减为 0，也不会从缓存老化掉。不经常被引用的对象将很快满足释放条件，在满足上面 3 个条件时将会被释放。

20.1.2　使用 T-SQL 查看执行计划

在 SQL Server 中，可以通过 SSMS 来显示估计的执行计划和实际的执行计划，也可以通过 T-SQL 语句中的 SET 选项来显示执行计划，另外还可以通过在 SQL Server Profiler 中设置相关的事件跟踪显示执行计划。

执行计划有文本格式（表格式）、XML 格式和图形化格式 3 种。文本格式的执行计划提供了层次结构以便于查看，XML 格式的执行计划一般用于程序读取，而图形化的执行计划可以最直观地显示执行计划的内容。

T-SQL 中提供了 SET 选项用于显示文本格式或者 XML 格式的执行计划。与执行计划相关的 SET 选项包括如下几种。

- SET SHOWPLAN_XML ON：此语句导致 SQL Server 不执行 T-SQL 语句。而 SQL Server 返回有关如何在正确的 XML 文档中执行语句的执行计划信息。
- SET SHOWPLAN_TEXT ON：执行该 SET 语句后，SQL Server 以文本格式返回每个查询的执行计划信息。不执行 Transact-SQL 语句或批处理。
- SET SHOWPLAN_ALL ON：该语句与 SET SHOWPLAN_TEXT 相似，但比 SHOWPLAN_TEXT 的输出格式更详细。
- SET STATISTICS XML ON：该语句执行后，除了返回常规结果集外，还返回每个语句的执行信息。输出正确的 XML 文档集。SET STATISTICS XML ON 为执行的每个语句生成一个 XML 输出文档。SET SHOWPLAN_XML ON 和 SET STATISTICS XML ON 的不同之处在于第二个 SET 选项执行 Transact-SQL 语句或批处理。SET STATISTICS XML ON 输出还包含有关各种操作符处理的实际行数和操作符的实际执行数。
- SET STATISTICS PROFILE ON：该语句执行后，除了返回常规结果集外，还返回每个语句的执行信息。两个 SET 语句选项都提供文本格式的输出。SET SHOWPLAN_ALL ON 和 SET STATISTICS PROFILE ON 的不同之处在于第二个

SET 选项执行 Transact-SQL 语句或批处理。SET STATISTICS PROFILE ON 输出还包含有关各种操作符处理的实际行数和操作符的实际执行数。

说明：除了这些设置选项外，还有 SET STATISTICS IO ON 用于显示 T-SQL 语句执行后生成的有关磁盘活动数量的信息。SET STATISTICS TIME ON 用于显示执行语句后，分析、编写和执行每个 T-SQL 语句所需的毫秒数。这两个选项在性能调优中尤为重要，但与执行计划无关所以并未列出。

例如要查询 AdventureWorks 2012 数据库中的 Person.AddressType 表，找出其中 Name 列为 Home 的行，同时以文本格式显示执行计划，则对应的 SQL 脚本如代码 20.1 所示。

代码 20.1　查询数据并显示执行计划

```
SET STATISTICS PROFILE ON --打开文本执行计划
SELECT *
FROM Person.AddressType
WHERE NAME='Home'
```

系统将返回查询结果，同时以表的形式显示执行计划，如图 20.1 所示。

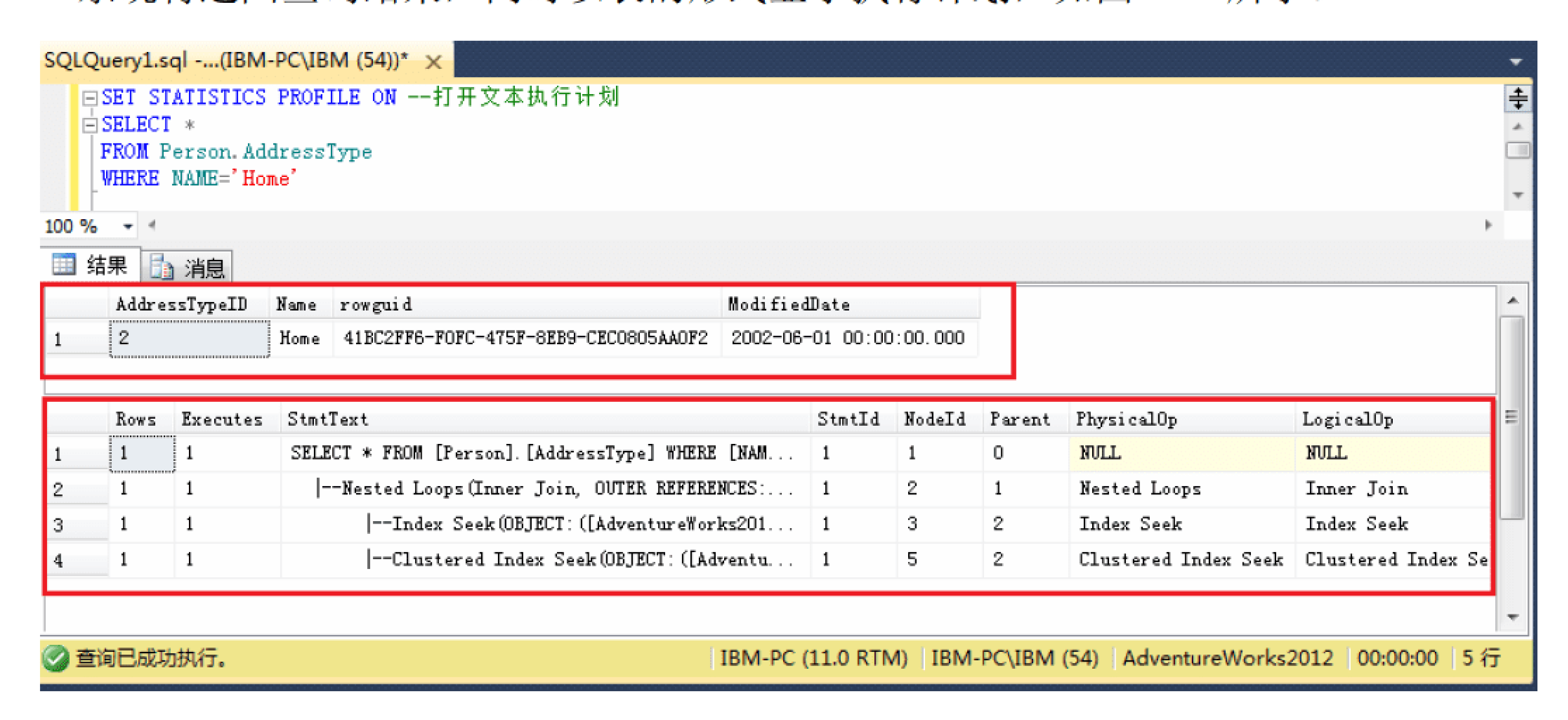


图 20.1　显示执行结果和执行计划

注意：SET 选项与会话相关，也就是说一旦执行了 SET 选项，该会话以后执行的所有 SQL 查询都将应用该选项，而其他会话则不受该 SET 选项的影响。

20.1.3　使用 SSMS 图形显示执行计划

在 SSMS 中输入查询的 T-SQL 语句，然后选择“查询”菜单下的“显示估计的执行计划”选项，或者使用快捷键 Ctrl+L，系统将不实际运行查询，而将该查询的执行计划以图形的方式显示出来，如图 20.2 所示。

如果既希望执行 SQL 语句，而且又要将执行计划显示出来，则只需要选中“查询”菜单中的“包括实际的执行计划”选项，然后运行该查询，系统将在结果选项卡和消息选项

卡旁新建“执行计划”选项卡，并以图形化的方式显示实际的执行计划。



图 20.2　显示估计的执行计划

对于在 T-SQL 中通过 SET SHOWPLAN_XML ON 等语句返回的 XML 类型的执行计划，也可以通过 SSMS 以图形化的方式显示出来。只需要将 XML 文件保存为.sqlplan 扩展名即可。

在图形化显示的执行计划中，除了显示执行计划中每个步骤的操作、开销外，再将鼠标指针悬停在某个步骤上，系统将显示该步骤的行数、估计 I/O 开销、估计 CPU 开销等更多详细的信息，如图 20.3 所示。

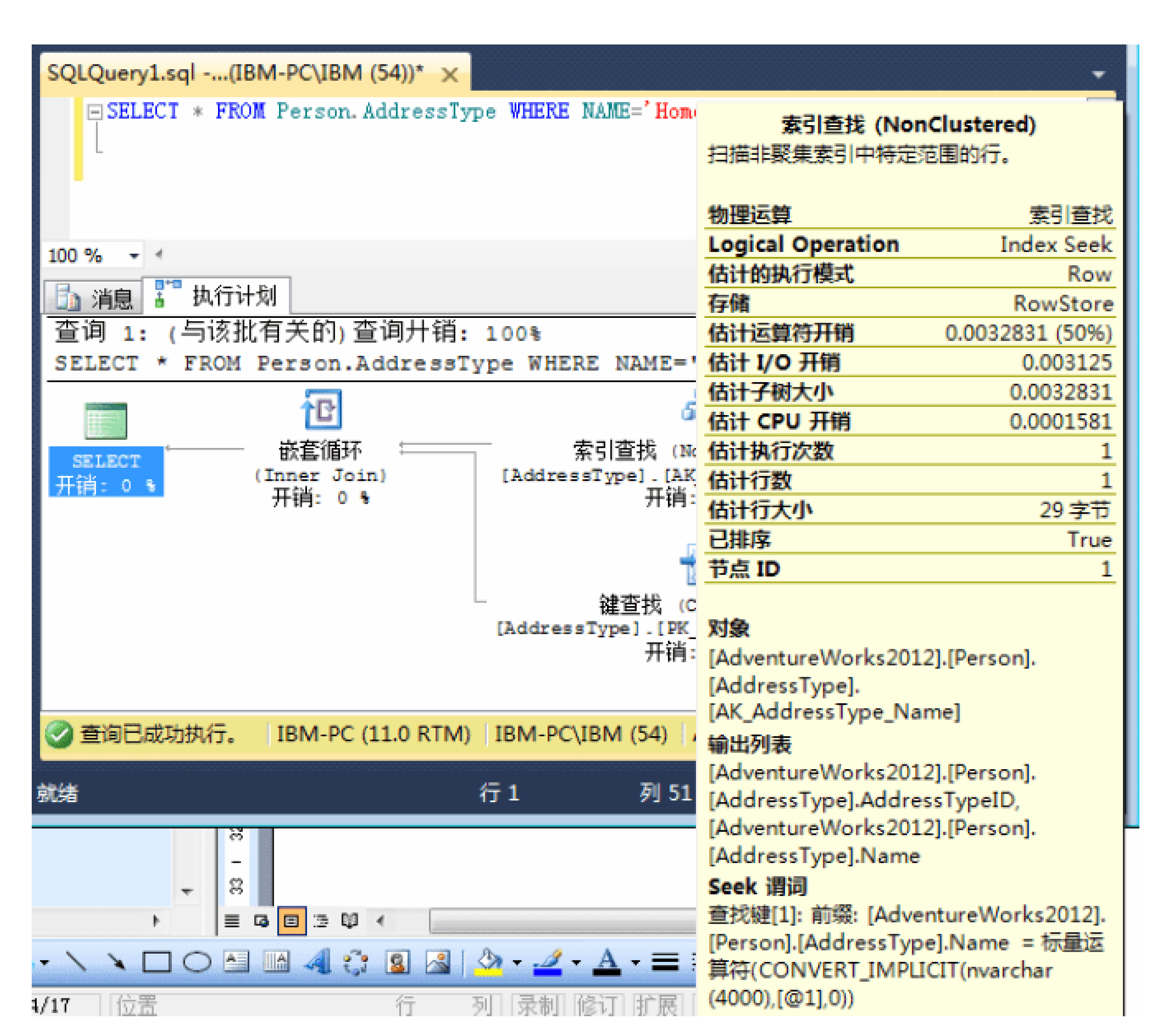


图 20.3　执行计划步骤的详细信息

在性能优化过程中，执行计划中重点关注的是执行的操作步骤类型和开销所在百分比，改进执行计划的步骤、优化开销较大的步骤将有助于性能的提升。

20.1.4　重新编译执行计划

前面已经讲到，SQL 语句、存储过程等执行时查看缓存中是否有现成的执行计划，如果有则直接使用，如果没有，则编译 SQL 语句和存储过程为执行计划进行执行，同时将该执行计划缓存起来。

在有些情况下，对于同一个存储过程，传入的参数不同，最优的执行计划也不相同，但是系统默认会从缓存中获取原有的执行计划。由于使用了错误的执行计划，将导致执行效率降低。对于这种情况，SQL Server 提供了 WITH RECOMPILE 选项用于指定不缓存执行计划，每次执行时都重新生成执行计划。例如现在有一个数据分布不均的表，其表的建立脚本如代码 20.2 所示。

代码 20.2　创建数据分布不均的表和数据

```
USE tempdb;
GO
CREATE TABLE t1 --创建测试表
(
    id INT IDENTITY PRIMARY KEY,
    NAME NVARCHAR(10) NOT NULL,
    CreateTime DATETIME DEFAULT(GETDATE())
)
GO
--接下来插入测试数据
SET NOCOUNT ON
INSERT INTO t1(NAME) VALUES(N'Hello')
DECLARE @i INT=0
WHILE @i<1000
BEGIN
    INSERT INTO t1(NAME) VALUES(N'Same')
    SET @i+=1
END
INSERT INTO t1(NAME) VALUES(N'TheEND')
CREATE INDEX IX_Name ON t1([Name])     --创建索引
```

接下来创建存储过程用于根据 Name 读取数据，对应的脚本如代码 20.3 所示。

代码 20.3　创建存储过程

```
CREATE PROC sp1     --测试用存储过程
@name NVARCHAR(10)
AS
SELECT *
FROM t1
WHERE [NAME]=@name
```

现在执行该存储过程，第一次传入只有一行数据的参数 TheEND，这时系统将会把存储过程编译成执行计划并缓存起来。第二次传入有很多行数据的参数 Same，系统将根据缓存中存储过程的执行计划进行执行，如图 20.4 所示。

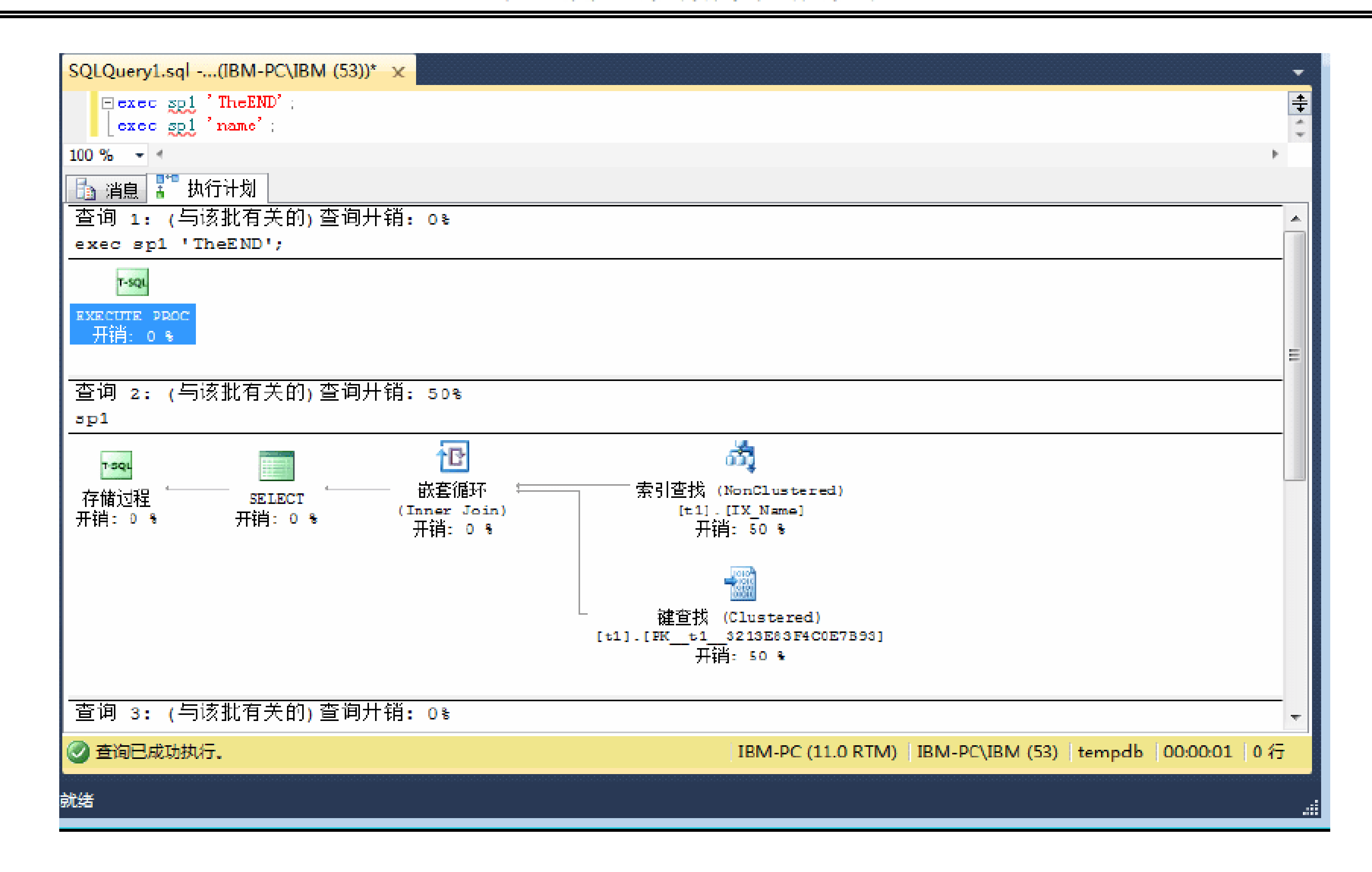


图 20.4　执行存储过程并显示执行计划

但是如果单独执行 SQL 语句查询 Name 为 Same 的数据，可以看到执行计划并不是像存储过程中执行那样，而是采用聚集索引扫描，将 I/O 统计打开可以看到明细的区别，如代码 20.4 所示。

代码 20.4　存储过程与 SQL 语句分别执行

```
SET STATISTICS IO ON    --打开 I/O 统计
exec sp1 'Same'
SELECT *
FROM t1
WHERE [NAME]='Same'
```

系统返回的统计信息：

```
表 't1'。扫描计数 1，逻辑读取 2005 次，物理读取 0 次，预读 0 次，lob 逻辑读取 0 次，
lob 物理读取 0 次，lob 预读 0 次。
表 't1'。扫描计数 1，逻辑读取 7 次，物理读取 0 次，预读 0 次，lob 逻辑读取 0 次，lob
物理读取 0 次，lob 预读 0 次。
```

本应该是聚集索引扫描，逻辑读取 7 次的，却因为使用了缓存中的执行计划，造成了逻辑读取 2005 次。如果数据量达到几十万、几百万时，这个差别将是惊人的。

修改存储过程定义，添加 WITH RECOMPILE 选项，系统将在每次执行时重新编译，具体修改和执行存储过程的脚本如代码 20.5 所示。

代码 20.5　每次重新编译存储过程

```
ALTER PROC sp1
@name NVARCHAR(10) WITH RECOMPILE --每次执行都重新编译
AS
SELECT *
FROM t1
WHERE [NAME]=@name
GO
exec sp1 'TheEND'
exec sp1 'Same'
```

执行该查询并打开执行计划，将可以看到这两个语句被编译成了不同的执行计划，如图 20.5 所示。

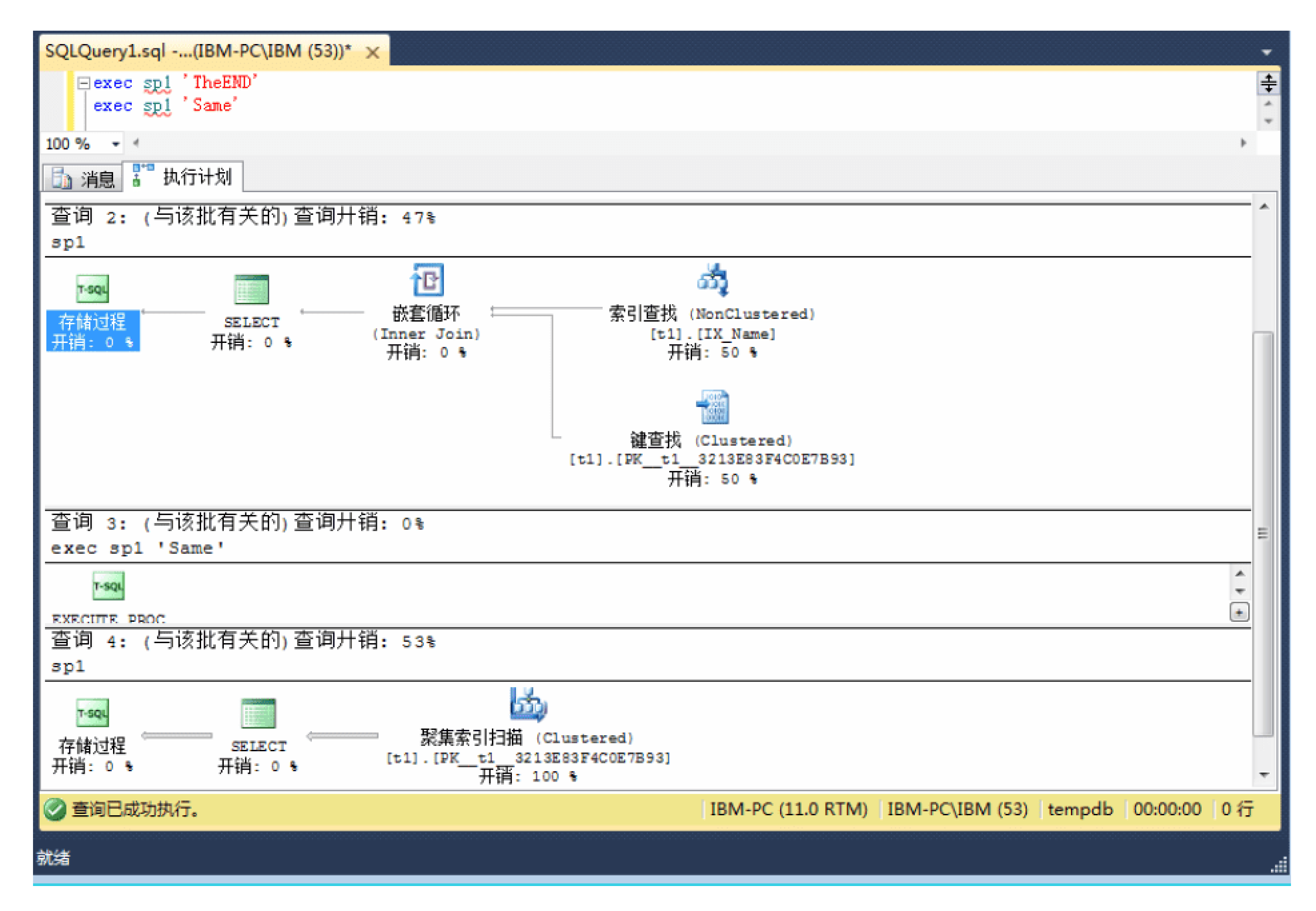


图 20.5　重新编译后的执行计划

如果不希望每次都重新编译执行计划，而是在希望重新编译的时候才重新编译，则不需要修改存储过程的定义，而是在调用存储过程时指定 WITH RECOMPILE 选项。例如不重新编译 TheEND 参数的执行计划，而重新编译 Same 的执行计划如代码 20.6 所示。

代码 20.6　指定重新编译执行计划

```
EXEC sp1 'TheEND'
EXEC sp1 'Same' WITH RECOMPILE
```

20.2　联　　接

联接（Join）是将两个表合并为一个表的操作。在前面介绍 SQL 基础时讲到联接分为外联接、内联接和交叉联接。本节将主要通过查询计划的角度从本质上讲解联接。

20.2.1　嵌套循环联接

嵌套循环联接（Nested Loop Join）也称为“嵌套迭代”，它将一个联接输入用作外部输入表（显示为图形执行计划中的顶端输入），将另一个联接输入用作内部（底端）输入表。外部循环逐行处理外部输入表。内部循环会针对每个外部行执行，在内部输入表中搜索匹配行。

最简单的情况是，搜索时扫描整个表或索引，这称为“单纯嵌套循环联接”。如果搜索时使用索引，则称为“索引嵌套循环联接”。如果将索引生成为查询计划的一部分（并

在查询完成后立即将索引破坏），则称为“临时索引嵌套循环联接”。查询优化器考虑了所有这些不同的情况。

如果外部输入较小而内部输入较大，且预先创建了索引，则使用嵌套循环联接尤其有效。在许多小事务中（如那些只影响较小的一组行的事务），索引嵌套循环联接优于合并联接和哈希联接。但在大型查询中，嵌套循环联接通常不是最佳选择。

在程序中可以理解为嵌套的 for 循环语句，先是对外部输入表循环找到每行数据，再使用循环对内部输入表的每行数据与外部输入表的数据进行匹配，直到这 2 个循环都完成。例如创建班级表和学生表，然后对这两个表进行内联接，则系统将采用嵌套循环联接对这两个表进行处理。创建表的脚本如代码 20.7 所示。

代码 20.7　创建班级和学生表

```
CREATE TABLE Class  --创建测试表
(
    CID INT IDENTITY PRIMARY KEY,
    CName VARCHAR(10) NOT NULL
)
GO
CREATE TABLE Student
(
    SID INT IDENTITY PRIMARY KEY,
    CID INT NOT NULL ,
    SName VARCHAR(10) NOT NULL,
    CONSTRAINT FK_Student_Class FOREIGN KEY(CID)  REFERENCES  Class(CID)
)
--插入测试数据
INSERT INTO Class VALUES(1,'01')
INSERT INTO Class VALUES(2,’02')
INSERT INTO Student (CID,SName) VALUES(1,'s11')
INSERT INTO Student (CID,SName) VALUES(1,'s12')
INSERT INTO Student (CID,SName) VALUES(2,'s21')
INSERT INTO Student (CID,SName) VALUES(2,'s22')
INSERT INTO Student (CID,SName) VALUES(2,'s23')
```

内联接查询这 2 个表，同时返回查询的执行计划，查询脚本如代码 20.8 所示，返回的执行计划如图 20.6 所示。

代码 20.8　使用内联接查询创建班级和学生表

```
SELECT *
FROM Student s
INNER JOIN Class c      --内联接
ON s.CID=c.CID
```

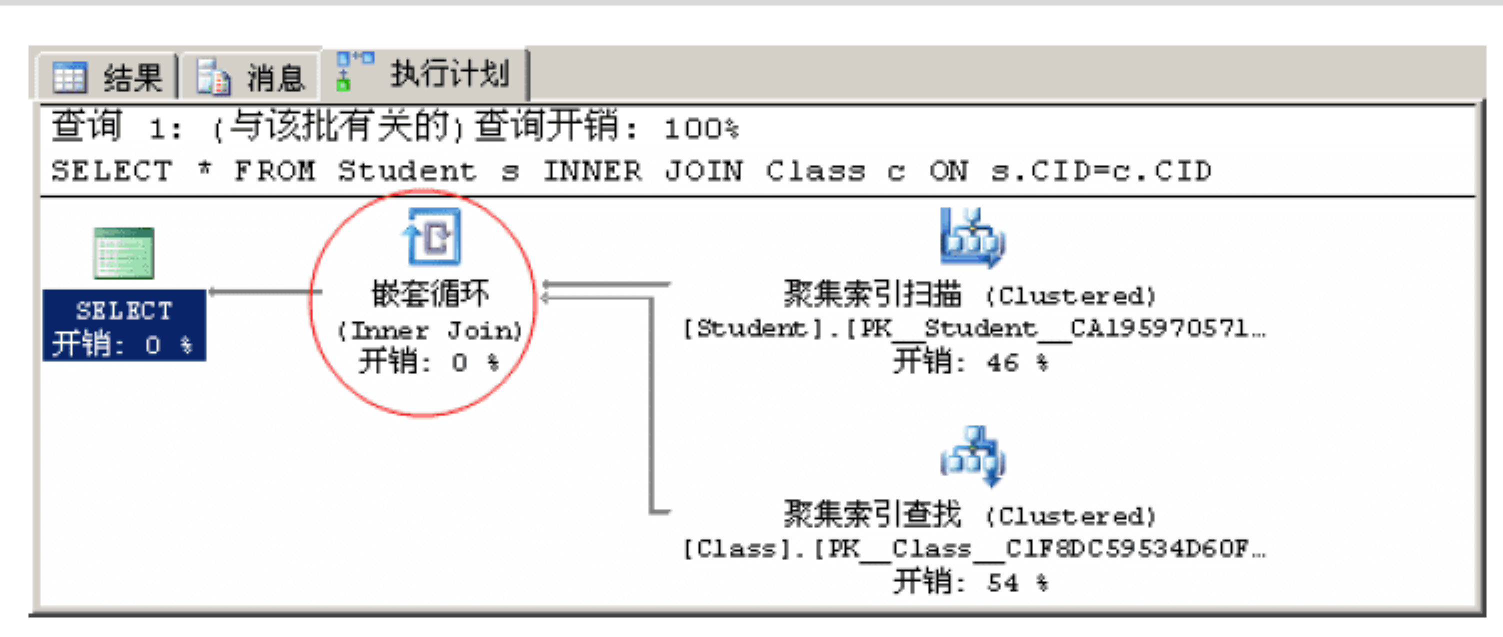


图 20.6　嵌套循环的执行计划

20.2.2　合并联接

合并联接（Merge Join）要求两个输入都在合并列上排序，其由联接谓词的等效（ON）子句定义。通常，利用查询优化器扫描索引（如果在适当的一组列上存在索引），或在合并联接的下面放一个排序运算符。在极少数情况下，虽然可能有多个等效子句，但只用其中一些可用的等效子句获得合并列。

由于每个输入都已排序，因此 Merge Join 运算符将从每个输入获取一行并将其进行比较。例如，对于内联接操作（INNER. JOIN），如果获取的行相等则返回该行。如果行不相等，则抛弃值较小的行并从该输入获得另一行进行比较。这一过程将重复进行，直到处理完所有的行为止。

合并联接操作可以是常规操作，也可以是多对多操作。多对多合并联接使用临时表存储行。如果每个输入中有重复值，则在处理其中一个输入中的每个重复项时，另一个输入必须重绕到重复项的开始位置。

如果存在驻留谓词，则所有满足合并谓词的行都将对该驻留谓词取值，而只返回那些满足该驻留谓词的行。

合并联接本身的速度很快，但如果需要执行排序操作，选择合并联接就会非常费时。然而，如果数据量很大且能够从现有 B 树索引中获得预排序的所需数据，则合并联接通常是最快的可用联接算法。例如仍然使用前面创建的班级和学生表，由于合并联接在两表数据量并不小而且联接列已排序的情况下发生，所以需要向表中添加大量数据，同时还要为进行联接的列 CID 排序，具体 SQL 脚本如代码 20.9 所示。

代码 20.9　为班级表和学生表添加数据

```
SET NOCOUNT ON
--创建大量的测试数据
DECLARE @i INT=3
WHILE @i<1000
BEGIN
    INSERT INTO Class VALUES(@i,'01')
    DECLARE @j int=0
    WHILE @j<10
    BEGIN
        INSERT INTO Student (CID,SName) VALUES(@i,'s'+CONVERT(VARCHAR
        (5),@i))
        SET @j+=1
    END
    SET @i+=1
END
GO
CREATE INDEX IX Student CID    --创建索引
ON Student(CID) include(SID,SName)
```

接下来再运行两表的联接查询，可以看到执行计划中使用了合并联接，如图 20.7 所示。

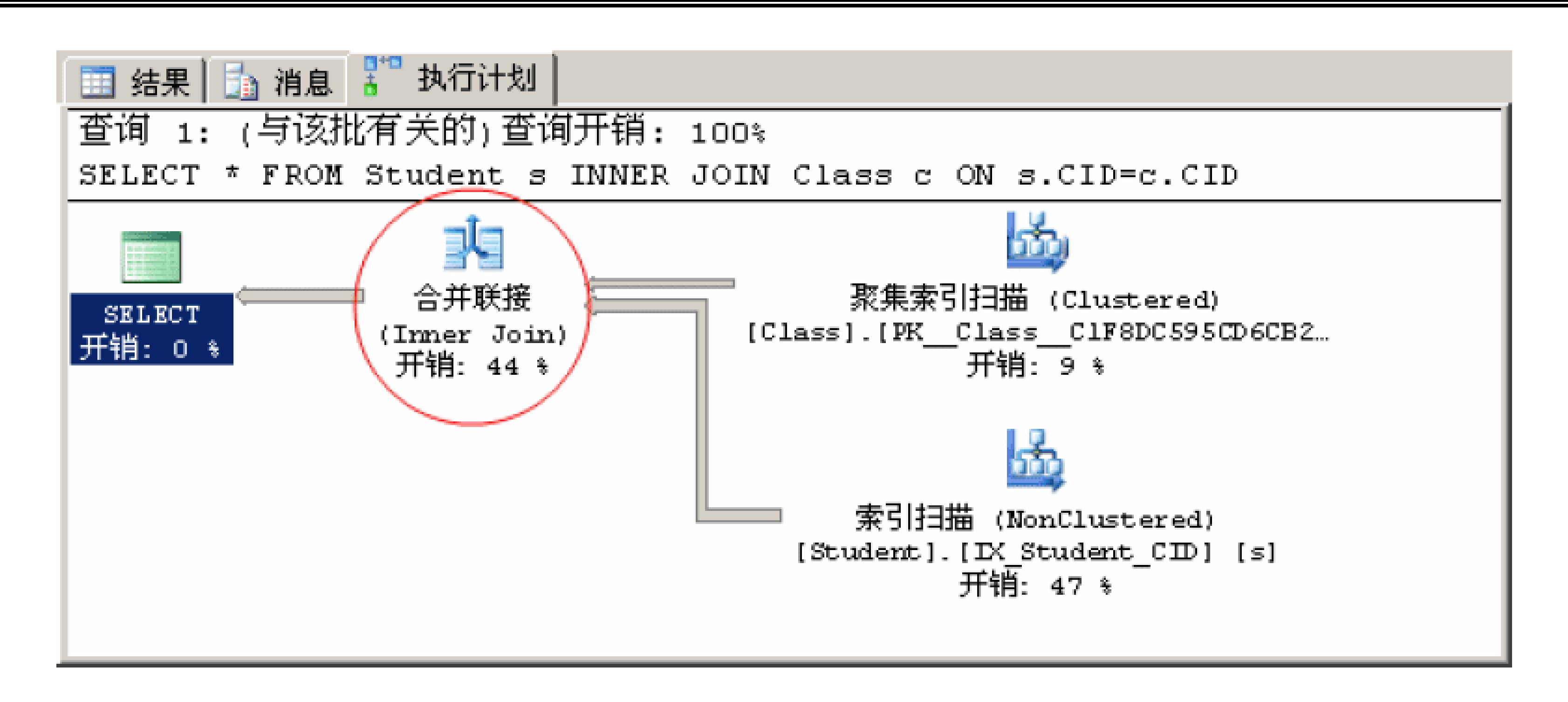


图 20.7　合并联接

20.2.3　哈希联接

哈希联接（Hash Join）有两种输入，即生成输入和探测输入。如果两个联接输入都很大，而且大小差不多，则预先排序的合并联接提供的性能与哈希联接相近。但是，如果这两个输入的大小相差很大，则哈希联接操作通常快得多。

哈希联接先扫描或计算整个生成输入，然后在内存中生成哈希表。根据计算得出的哈希键的哈希值，将每行插入哈希存储桶。如果整个生成输入小于可用内存，则可以将所有行都插入哈希表中。生成阶段之后是探测阶段。一次一行地对整个探测输入进行扫描或计算，并为每个探测行计算哈希键的值，扫描相应的哈希存储桶并生成匹配项。

哈希联接一般在一张小表和一张大表进行联接时应用。例如对于班级表和学生表，明显班级表要比学生表小很多，在去掉学生表上对 CID 的排序后，这两个表在进行联接运算时将使用哈希联接。去掉 CID 的排序和联接查询如代码 20.10 所示。

代码 20.10　去掉排序然后联接查询

```
DROP INDEX IX_Student_CID ON Student   --去掉索引
GO
SELECT *
FROM Student s
INNER JOIN Class c
ON s.CID=c.CID
```

查询出的执行计划如图 20.8 所示。

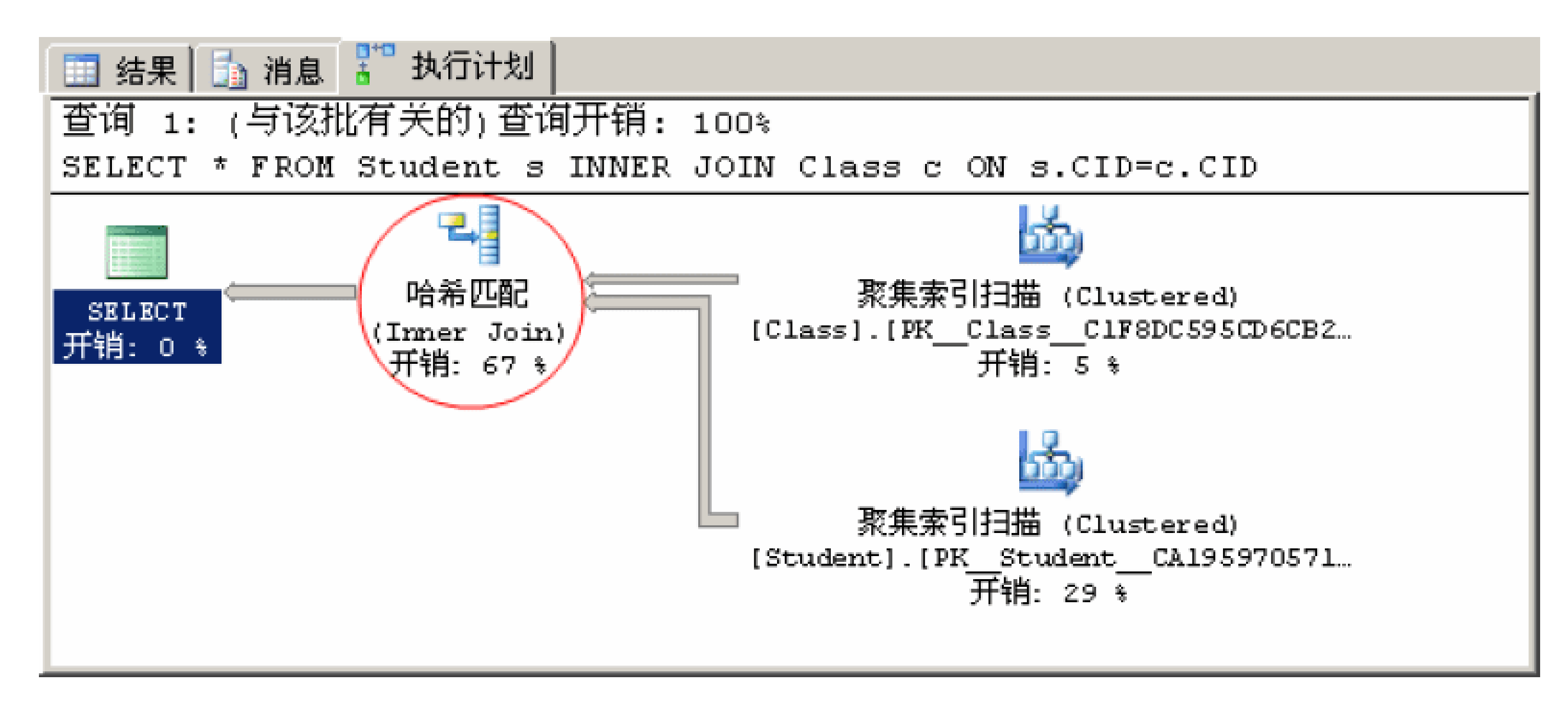


图 20.8　哈希联接

20.3　SARG 查询参数

编写 SQL 查询时经常需要使用 WHERE 子句后跟查询参数，不正确地编写查询参数将会导致索引无法使用，造成系统性能下降，本节将主要讲解编写查询参数的注意事项。

20.3.1　SARG 简介

查询参数 SARG 是 Search ARGument 的简称，只有符合 SARG 的查询，才能由系统建立有效使用索引的执行计划。

SARG 中定义如果是一个条件的查找，则查找限制为完全符合或一个范围的值，如果是多个条件的查找则以 AND 连接查找的条件。SARG 包含常量表达式来与数据表中的字段做比较。字段名称出现在操作的一边，而常量或变量出现在另一边。如果字段名称同时出现在操作的两边就不算 SARG。SARG 可以包含以下操作：=、<、>、>=、<=、BETWEEN 以及部分 LIKE。LIKE 中如果%不是出现在前面则符合 SARG。例如 LIKE '梁%'就符合 SARG，而 LIKE'%恒'则不符合 SARG。

SARG 在查询中代表用来查找的常量或变量可以直接与索引键做比较。对于非 SARG 语句，SQL Server 将无法使用索引。非 SARG 语句一般包含下列操作：NOT、!=、<>、!>、!<、NOT EXISTS、NOT IN 和 NOT LIKE 等。另外，使用 LIKE '%abc%'将会导致全表扫描，降低性能。

20.3.2　在查询中使用 SARG

在数据查询中不要对数据列做运算，否则将不符合 SARG。例如在 AdventureWorks 2012 数据库中，若要查询人员 Rob Walters 的信息，则对应的正确且符合 SARG 的 SQL 查询如代码 20.11 所示。

代码 20.11　符合 SARG 的查询

```
SELECT *
FROM Person.Person
WHERE FirstName='Rob' AND LastName='Walters'
```

而如果对查询的列做运算，则不符合 SARG，如代码 20.12 所示。

代码 20.12　不符合 SARG 的查询

```
SELECT *
FROM Person.Person
WHERE FirstName+' '+LastName='Rob Walters' --写法错误
```

这里符合 SARG 的查询将可以使用索引进行查找，而非 SARG 则不能使用索引，只能进行聚集索引扫描（相当于扫描整个表），它们的执行计划如图 20.9 所示，非 SARG 下的表扫描占用了 100%的开销，而 SARG 查询占用 0%的开销。

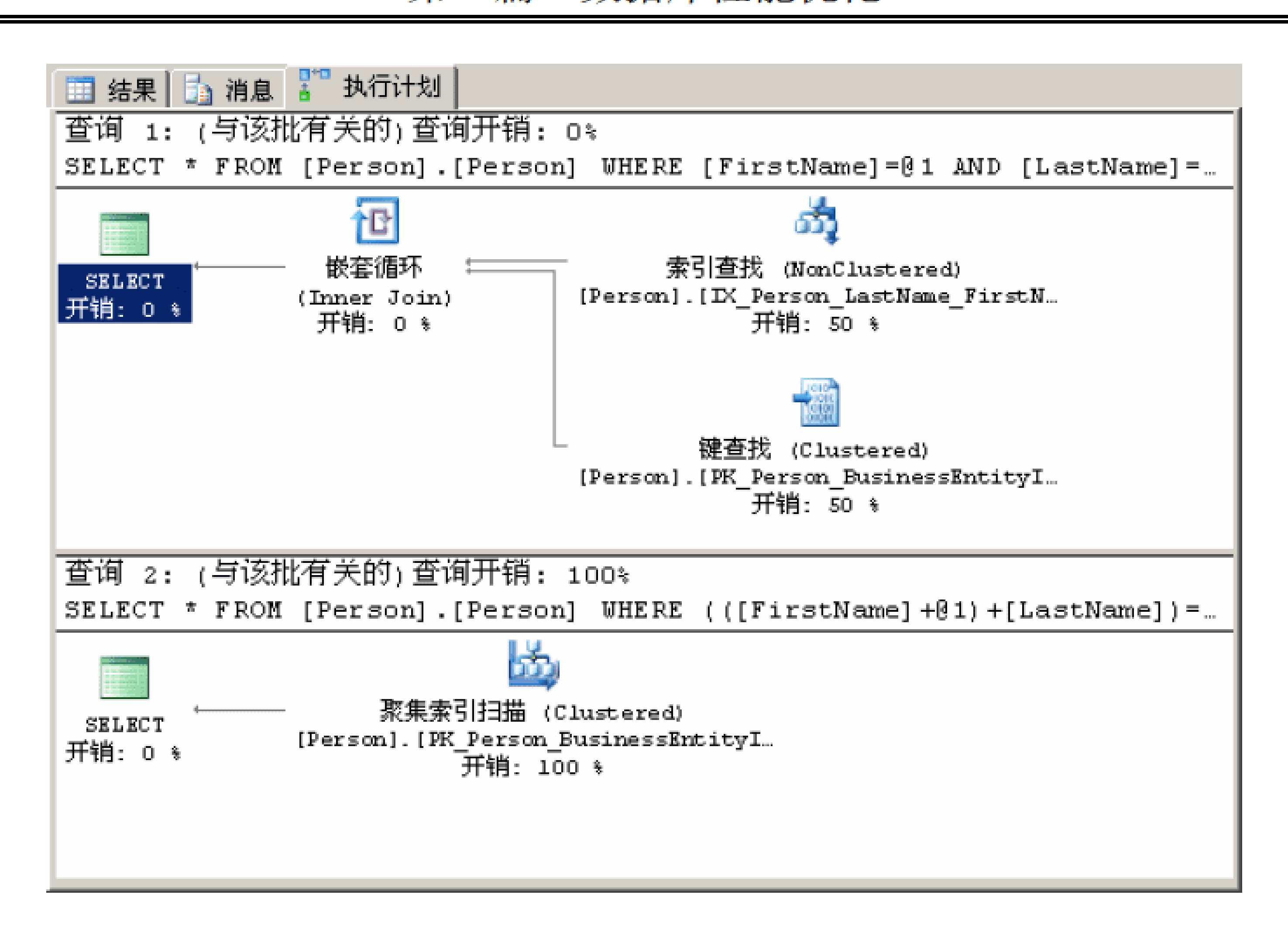


图 20.9　SARG 和非 SARG 的区别

在编写查询语句时尽量不要使用负向查询。因为通过索引的顺序结构，SQL Server 可以利用二分查找快速地找到对应的数据，但是若使用负向查询，则无法利用索引进行二分查找，只能进行表的扫描。

注意：SQL Server 2012 中加强了对查询意图的分析，所以大部分情况下写成 A>5 OR A<5 与写成 A!=5 是没有区别的，SQL Server 都能分析出查询的意图而利用索引。

除了不应该对列做运算外，也不能在 WHERE 子句中对字段使用函数。如果使用函数，则 SQL Server 需要将数据表内所有记录的相关字段输入到函数中，如果有 1000 万条记录，则需要执行 1000 万次函数的调用，其执行时的性能自然很低。

例如要从客户表中找到 AccountNumber 以 AW0003011 开头的数据，如果使用函数 LEFT()对 AccountNumber 进行运算，然后与“AW0003011”匹配，则该查询不符合 SARG，将无法使用该表中的索引。可以使用 SARG 的 LIKE 操作代替 LEFT( )函数，两个 SQL 查询脚本如代码 20.13 所示。

代码 20.13　非 SARG 和 SARG 的查询

```
SELECT *
FROM Sales.Customer
WHERE LEFT(AccountNumber,9)='AW0003011'     --不符合 SARG
SELECT *
FROM Sales.Customer
WHERE AccountNumber LIKE 'AW0003011%'       --符合 SARG
```

对应的执行计划如图 20.10 所示，非 SARG 花费了 91%的开销，而 SARG 只花费了 9%的开销。

在查询中，如果使用 OR 操作，多个条件中若有一个字段没有合适的索引，则其他再多的字段有索引也没有用，只有将整个表扫描一遍，以确定全部的数据是否有符合的记录。所以在查询中应该尽量避免使用 OR 操作，如果要使用，则一定要检查每个条件中都能够用到索引，否则将不是 SARG 操作。

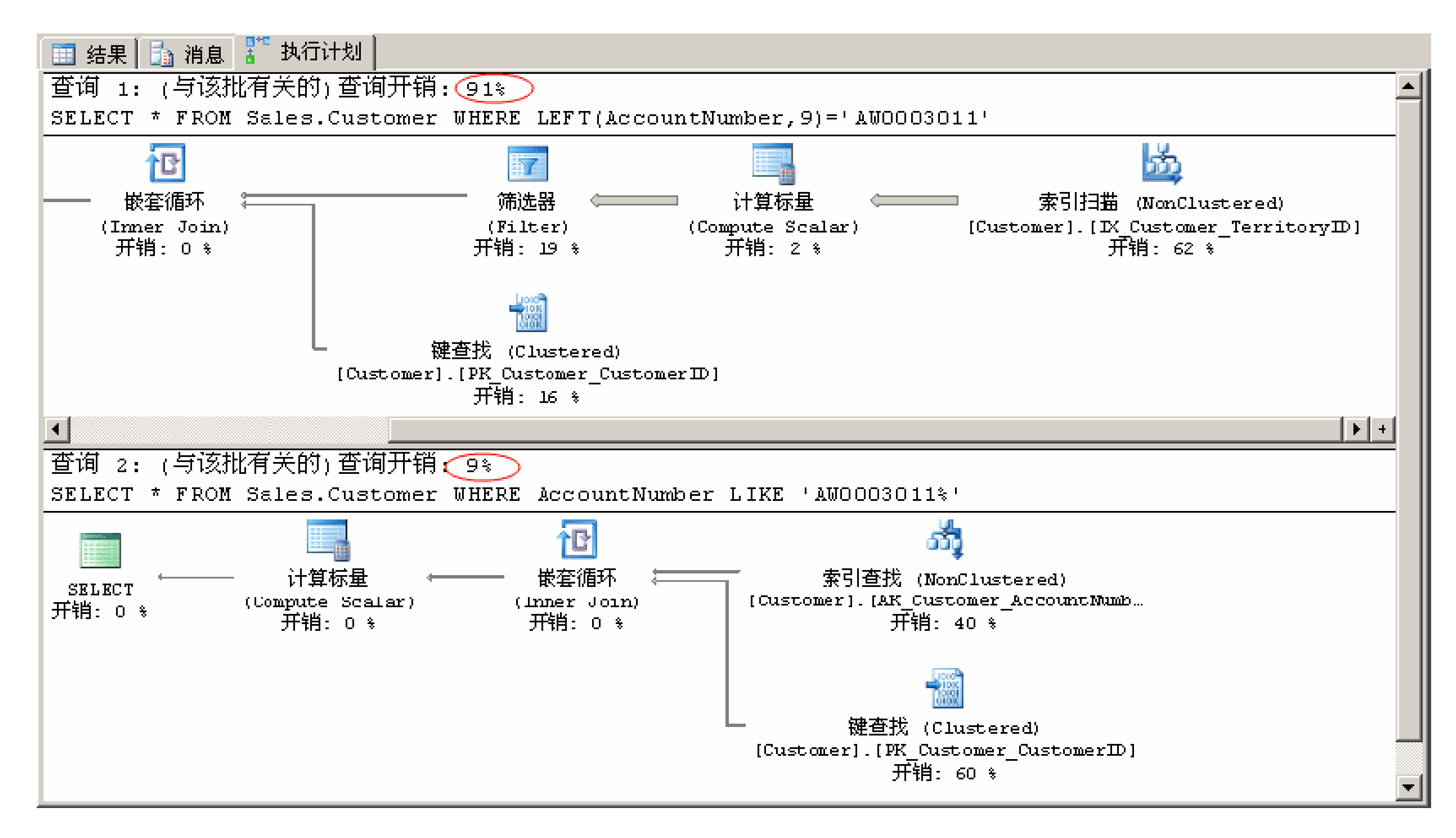


图 20.10　非 SARG 和 SARG 的执行计划比较

20.4　统 计 信 息

SQL Server 在得到 SQL 语句后可以将语句编译成不同的执行计划，而每个执行计划实际执行效率不一样，在执行之前，SQL Server 就通过统计信息来判断哪个执行计划最优，从而选择最优的执行计划。

20.4.1　统计信息简介

SQL Server 需要通过统计信息来判断哪个执行计划成本更小，以选择最优的执行计划，所以 SQL Server 将自动创建并维护有关列中值的分布情况的统计信息。创建统计信息后，数据库引擎对列值进行排序，并根据这些值创建一个“直方图”。直方图指定有多少行精确匹配每个间隔值，有多少行在间隔范围内，以及间隔中值的密度大小或重复值的发生率。

除了直方图中的信息外，其他信息是通过在字符串类型的列上创建的统计信息收集的，这些信息称为“字符串摘要”。字符串摘要可以帮助查询优化器在估计字符串模式中查询谓词的选择性，当查询中有 LIKE 条件时，使用字符串摘要可以更准确地估计结果集大小，并不断优化查询计划。

由于查询一般要使用到索引，所以在创建索引时，系统将自动存储有关索引列的统计信息。另外，当 AUTO_CREATE_STATISTICS 数据库选项设置为 ON（默认值）时，数据库引擎自动为没有用于谓词的索引列创建统计信息。

统计信息和索引一样，会随着列中数据的变化而变化。如果数据发生变化后统计信息没有更新，则可能导致查询优化器选择查询的执行计划时不能选择最佳的。当 AUTO_UPDATE_STATISTICS 数据库选项设置为 ON（系统默认就是 ON 值）时，查询优化器会在表中的数据发生变化时自动定期更新这些统计信息。

在统计信息更新时，系统将对数据进行采样分析，采样是在各个数据页上随机进行的，取自表或统计信息所需列的最小非聚集索引。从磁盘读取一个数据页后，该数据页上的所有行都被用来更新统计信息。一般情况下，在大约有 20%的数据行发生变化时系统将更新统计

信息。更新时查询优化器会确保采样的行数尽量少。但是对于小于 8MB 的小表，由于它们数据量不大，完整采样并不会特别消耗资源，所以始终对其进行完整扫描来收集统计信息。

使用采样分析数据的方式可以将统计信息自动更新的开销降至最低。在某些情况下，统计采样无法获得表中数据的精确特征，这种情况下可以使用 UPDATESTATISTICS 语句的 SAMPLE 和 FULLSCAN 子句，控制按逐个表的方式手动更新统计信息时采样的数据量。FULLSCAN 子句指定扫描表中的所有数据来收集统计信息，而 SAMPLE 子句用来指定采样的行数或行数百分比。

20.4.2　使用 T-SQL 创建统计信息

默认情况下，统计信息都是由系统自动创建和更新的，用户并不需要干预统计信息的管理。但是对于一些特殊的情况，仍然需要人为地管理统计信息，比如禁用统计信息的自动功能、创建新的统计信息、手动更新统计信息等。

创建统计信息最直接的方法是使用系统存储过程 sp_createstats，使用该存储过程可以对当前数据库中所有用户表中适于统计的列创建统计信息。该存储过程的语法如代码 20.14 所示。

代码 20.14　sp_createstats 的语法

```
sp createstats [ [ @indexonly = ] 'indexonly' ]
       [ , [ @fullscan = ] 'fullscan' ]
    [ , [ @norecompute = ] 'norecompute' ]
```

其中第 1 个参数[@indexonly=]'indexonly'指定创建统计信息时只应考虑参与索引的列。默认值为 NO。第 2 个参数[@fullscan=]'fullscan'指定将 FULLSCAN 选项用于 CREATE STATISTICS。如果忽略 fullscan，SQL Server 数据库引擎将执行默认示例扫描，默认值为 NO。第 3 个参数[@norecompute=]'norecompute'指定对新创建的统计信息禁用统计信息自动重新计算功能，默认值为 NO。

例如，要给 TestDB1 中的所有表中创建了索引的列创建统计信息，而且使用全表扫描的方式获得统计信息，则对应的语句为：

```
sp_createstats @indexonly='indexonly',@fullscan='fullscan'
```

注意：系统存储过程 sp_createstats 只是创建统计信息，对于已经存在的统计信息将不会受到影响。

除了使用 sp_createstats 自动创建统计信息外，还可以使用 CREATE STATISTICS 语句对特定表或视图列创建统计信息，并使用 UPDATE STATISTICS 语句更新统计信息。对与索引无关的表或视图，可以创建的最大统计信息数为 2000。对于任何可以作为索引键的列或列的组合，都可以创建统计信息，对于大型对象类型列和组合列值最大大小可以超过 900 字节的字段也可以创建统计信息。CREATE STATISTICS 的语法如代码 20.15 所示。

代码 20.15　CREATE STATISTICS 的语法

```
CREATE STATISTICS statistics_name
ON { table | view } ( column [ ,...n ] )
    [ WHERE <filter_predicate> ]
    [ WITH
```

```
        [ [ FULLSCAN
          | SAMPLE number { PERCENT | ROWS }
          | STATS_STREAM = stats_stream ] [ , ] ]
        [ NORECOMPUTE ]
    ] ;
```

其中比较重要的几个参数的含义如下所示。

- statistics_name 是要创建的统计组的名称。
- table 为要在其中创建命名统计的表的名称。
- view 为要在其中创建命名统计的视图的名称。在视图中创建统计信息之前，视图必须有聚集索引。
- column：要在其中创建统计信息的一列或一组列。
- WHERE<filter_predicate>指定一个表达式，以选择要包含在筛选统计信息中的行。
- FULLSCAN 指定应读取 table 或 view 中的所有行以收集统计信息。指定 FULLSCAN 具有与 SAMPLE 100 PERCENT 相同的行为。此选项不能与 SAMPLE 选项一起使用。
- SAMPLE number{PERCENT|ROWS}指定通过随机抽样应读取的数据百分比或指定的数据行数，收集统计信息。number 必须为整数。如果指定了 PERCENT，则 number 应该在 0～100 之间；如果指定了 ROWS，则 number 可以在 0 至总行数 n 之间。
- NORECOMPUTE 指定数据库引擎不应自动重新计算统计信息。如果指定了该选项，那么即使数据发生更改，数据库引擎仍将继续使用以前创建的旧统计信息。

例如，在 AdventureWorks2012 数据库中为表 Employee 的 BirthDate 创建统计信息，使用全表扫描的方式创建，则对应的 SQL 脚本如代码 20.16 所示。

代码 20.16　创建统计信息

```
USE AdventureWorks2012;
GO
CREATE STATISTICS ST_Employee_BirthDate      --统计信息名
ON HumanResources.Employee(BirthDate)        --创建统计信息的表和字段
WITH FULLSCAN --全表扫描
```

20.4.3　使用 T-SQL 管理统计信息

默认情况下，统计信息在创建后系统会自动更新其中的内容，但是我们仍然有可能需要手动去管理它们。使用 UPDATE STATISTICS 命令即可更新统计信息，其语法如代码 20.17 所示。

代码 20.17　UPDATE STATISTICS 语法

```
UPDATE STATISTICS table | view
    [
        {
            { index | statistics_name } | ( { index |statistics_name } [ ,...n ] )
        }
    ]
    [    WITH [
            [ FULLSCAN ]
```

```
            | SAMPLE number { PERCENT | ROWS } ]
            | RESAMPLE
            | <update stats stream option> [ ,...n ]
        ]
        [ [ , ] [ ALL | COLUMNS | INDEX ]
        [ [ , ] NORECOMPUTE ]
    ] ;
```

其中的参数与创建统计信息时的参数基本相同，在此不再详细说明。例如要更新前面创建的统计信息 ST_Employee_BirthDate，将使用 80%采样的方式进行统计，则对应的 SQL 脚本如代码 20.18 所示。

代码 20.18　更新统计信息

```
UPDATE STATISTICS HumanResources.Employee  --表名
ST_Employee_BirthDate                      --统计信息名
WITH SAMPLE 80 PERCENT                     --使用 80%的采样
```

如果不再需要保留和维护为一个列生成的统计信息，可以使用 DROP STATISTICS 命令删除。例如要将前面创建的统计信息 ST_Employee_BirthDate 删除，则对应的脚本为：

```
DROP STATISTICS  HumanResources.Employee.ST_Employee_BirthDate
```

20.4.4　使用 SSMS 创建和管理统计信息

使用 SSMS 可以快速，方便地管理统计信息。如果要新建统计信息，只需要在对象资源管理器中展开对应表节点下的“统计信息”节点，在弹出的快捷菜单中选择“新建统计信息”选项，将打开新建统计信息对话框，如图 20.11 所示。

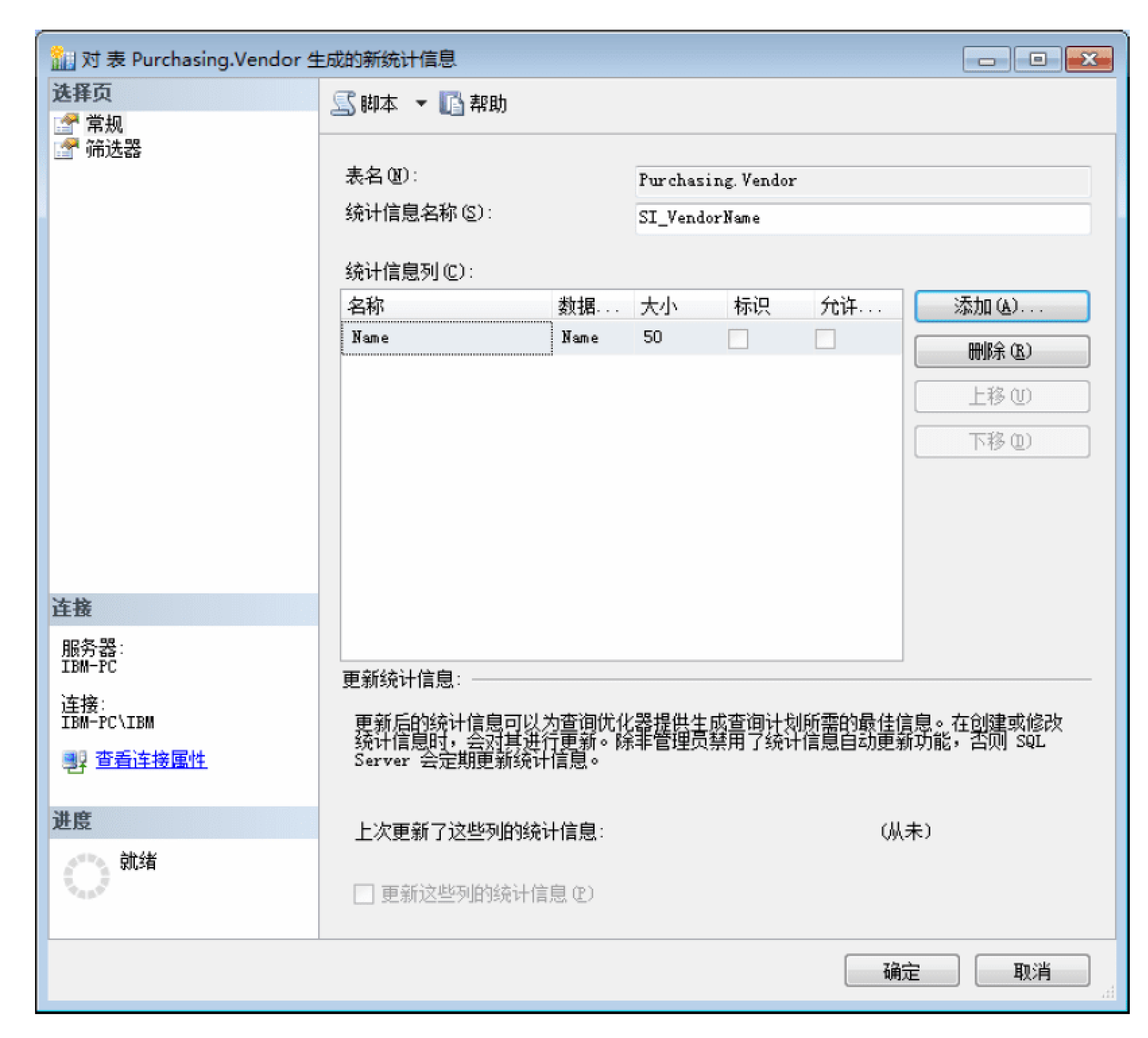


图 20.11　新建统计信息对话框

在“统计信息名称”文本框中输入统计信息的名称，单击“添加”按钮添加需要统计的列，然后单击“确定”按钮即可完成统计信息的创建。

更新统计信息时只需要打开对应统计信息的属性窗口，然后选中其中的“更新这些列的统计信息”复选框，再单击“确定”按钮即可。

删除统计信息则更简单，在 SSMS 中使用快捷键 Delete 即可完成统计信息的删除。

20.5　小　　结

本章主要讲解了数据查询中的执行计划和联接操作，另外还有在查询中要注意的查询参数和与执行计划紧密关联的统计信息。

每个要执行的 SQL 语句都将被编译成执行计划，然后再将执行计划传入数据库引擎中负责执行，SQL Server 将编译的执行计划缓存起来，在下次执行相同的语句时直接调用缓存中的执行计划，而不用再编译 SQL 语句。可以以图形、文本或 XML 的方式查看执行计划。在进行联接查询操作中，SQL Server 提供了嵌套循环联接、合并联接和哈希联接。在编写 WHERE 查询条件时，一定要符合 SARG，符合 SARG 的查询将有效地利用索引，提高查询的效率。一个查询可以被编译成多种不同的执行计划，查询优化器使用统计信息并通过估计使用索引评估查询的开销来确定最佳查询计划。

第 21 章　事 务 处 理

SQL Server 是一个多任务多用户的数据库系统，所以在同一个时间中将进行很多个数据库操作，SQL Server 使用锁和事务的机制来保证数据的一致性。本章将主要讲解与事务处理相关的知识。

21.1　事　　务

在对多个数据库对象执行顺序操作时，有时需要使用事务来保证整个操作过程的完整性和数据库的一致性。本节将主要讲解事务的基础知识和如何在 SQL Server 中使用事务。

21.1.1　事务概述

事务是单个的工作单元。在一个事务中可以定义多个数据修改操作，事务成功，则表示在该事务中进行的所有数据修改均会提交，成为数据库中的永久组成部分。如果事务操作中途遇到错误，且必须取消或回滚，则包括错误发生之前的数据修改在内，所有数据修改均会被取消。

事务是作为单个逻辑工作单元执行的一系列操作。一个逻辑工作单元必须有 4 个属性，称为原子性、一致性、隔离性和持久性（合称为 ACID）属性，只有这样才能成为一个事务。

- 原子性（Atomicity），也叫做不可部分完成性。事务必须是原子工作单元，对于其数据修改，要么全都执行，要么全都不执行，不可能只执行了其中的一部分。
- 一致性（Consistency），事务在完成时，必须使所有的数据都保持一致状态。在相关数据库中，所有规则都必须应用于事务的修改，以保持所有数据的完整性。所有的内部数据结构（如 B 树索引或双向链表）在事务结束时都必须是正确的。
- 隔离（Isolation），由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务识别数据时数据所处的状态，要么是另一并发事务修改它之前的状态，要么是第二个事务修改它之后的状态，事务不会识别中间状态的数据。这称为可串行性，因为它能够重新装载起始数据，并且重播一系列事务，以使数据结束时的状态与原始事务执行的状态相同。
- 持久性（Durability），事务完成之后，它对于系统的影响是永久性的。该修改即使出现系统故障也将一直保持。

数据库应用程序可以指定事务何时开始与结束，控制整个事务的运行。事务是基于连接的，一个连接中开始了一个事务后只要没有进行提交或回滚，则这个连接后面所有执行

的 SQL 操作都是事务的一部分。SQL Server 以下列事务模式运行：

- ❑ 自动提交事务，每条单独的语句都是一个事务。例如一句简单的 UPDATE 操作，可能一次就更新了所有的数据，也可能出现错误不更新任何数据。
- ❑ 显式事务，每个事务均以 BEGIN TRANSACTION 语句显式开始，以 COMMIT 或 ROLLBACK 语句显式结束。这种事务的开始和结束就是通过人为操作，以命令来完成。
- ❑ 隐式事务，不用明确的以 BEGIN TRANSACTION 语句来激活事务，在前一个事务完成时新事务隐式启动，但每个事务仍以 COMMIT 或 ROLLBACK 语句显式完成。在连接上启用隐式事务需要使用 SET IMPLICIT_TRANSACTIONS ON 命令。
- ❑ 批处理级事务，只能应用于多个活动结果集（Multiple Active Result Set，MARS），在 MARS 会话中启动的 T-SQL 显式或隐式事务变为批处理级事务。当批处理完成时没有提交或回滚的批处理级事务自动由 SQL Server 进行回滚。

事务在激活后可以有 3 种办法结束事务。一种是使用 COMMIT 命令提交事务更改，一种是使用 ROLLBACK 命令回滚事务更改，还有就是强行中断事务所在的连接。使用 KILL 命令可以中断指定的连接。中断连接后，连接中的事务将会回滚到事务开始前的状态。

注意： KILL 命令无法结束系统进程、执行扩展存储过程和自己所属的进程。

21.1.2　使用事务

在 SQL Server 中使用 BEGIN TRANSACTION 命令可以显式地启动一个事务，该命令的语法如代码 21.1 所示。

代码 21.1　BEGIN TRANSACTION 语法

```
BEGIN { TRAN | TRANSACTION }
    [ { transaction_name | @tran_name_variable }
      [ WITH MARK [ 'description' ] ]
    ]
```

其中，transaction_name 是分配给事务的名称。transaction_name 必须符合标识符规则，但标识符所包含的字符数不能大于 32。仅在最外面的 BEGIN…COMMIT 或 BEGIN…ROLLBACK 嵌套语句对中使用事务名。

@tran_name_variable 为用户定义的、含有有效事务名称的变量的名称。必须用 char、varchar、nchar 或 nvarchar 数据类型声明变量。如果传递给该变量的字符多于 32 个，则仅使用前面的 32 个字符，其余的字符将被截断。

WITH MARK['description']指定在日志中标记事务。description 是描述该标记的字符串。如果使用了 WITH MARK，则必须指定事务名。WITH MARK 允许将事务日志还原到命名标记。

BEGIN TRANSACTION 语句相当于对当前数据库创建了一个副本，接下来的数据库操作都是在这个副本上运行，而不是直接应用到实际数据库中。例如要启动一个事务 trans1，在该事务中对表 t1 插入一行数据，则对应的 SQL 脚本如代码 21.2 所示。

代码 21.2　启用事务

```
CREATE TABLE t1      --在启用事务之前先创建好表 t1
(
    c1 INT NOT NULL
)
GO
BEGIN TRAN trans1    --启用事务 trans1
INSERT INTO t1 (c1)
VALUES (100)
```

现在在 SSMS 中新打开一个查询窗口，使用 SELECT 命令查询 t1 表中的数据将无法查询成功，因为事务 trans1 没有完成，对表 t1 的操作没有提交。

前面已经介绍到，使用 COMMIT 或 ROLLBACK 命令可以结束当前的事务。COMMIT 命令用于提交从 BEGIN TRANSACTION 语句开始到 COMMIT 语句之间的数据库操作到实际的数据库中。提交后事务对数据库的更改就是永久性的，不能再进行回滚。COMMIT 命令的语法是：

```
COMMIT { TRAN | TRANSACTION } [ transaction_name | @tran_name_variable ] ]
```

对于前面启用的事务 trans1，若需要提交该事务，只需要在启用该事务的查询窗口中运行下面命令即可。

```
COMMIT TRAN trans1
```

事务一旦提交后，另一个查询窗口中对 t1 表的 SELECT 查询也立即完成，可以看到 INSERT 操作成功完成。

如果事务在启用后并没有使用 COMMIT 命令进行提交，这时若撤销事务中进行的数据更改则需要使用 ROLLBACK 命令，其语法格式如代码 21.3 所示。

代码 21.3　ROLLBACK 语法

```
ROLLBACK { TRAN | TRANSACTION }
     [ transaction_name | @tran_name_variable
     | savepoint_name | @savepoint_variable ]
```

其中，transaction_name 就是要回滚的事务名称。例如创建一个事务 trans2，在该事务中执行了对 t1 表的 INSERT 操作，然后再将该事务回滚，则对应的 SQL 脚本如代码 21.4 所示。

代码 21.4　回滚事务

```
BEGIN TRAN trans2        --开启事务 2
INSERT INTO t1 (c1)
VALUES (50)
--以下回滚事务
ROLLBACK TRAN trans2
```

现在再到另一个查询窗口中查询 t1 表，将发现 t1 表中并不存在 50 这行数据。

注意：事务的命名并不是必须的，即使在 BEGIN TRANSACTION 语句中对事务进行了命名，但是在提交或回滚事务时也可以不使用事务名，直接利用 COMMIT 或 ROLLBACK 命令即可。

21.1.3　嵌套事务

SQL Server 支持对显式事务进行嵌套。在嵌套事务中，SQL Server 将忽略内部的所有事务的提交，根据最外部事务是提交还是回滚来决定内部事务的提交和回滚。如果提交了外部事务，则也会提交内部事务；如果回滚了外部事务，也会回滚所有内部事务。

在嵌套事务中，事务的提交和回滚不能越级进行。也就是说，在内部事务中，即使在 COMMIT 命令中引用外部事务的名称来提交外部事务，但该提交仍然对应的是内部的事务。

对于回滚事务，ROLLBACK 命令后不能跟内部事务的名称，只能跟最外部事务的名称。不管是否使用了事务名称，ROLLBACK 命令都将会回滚所有层级的事务，而不仅仅是当前层的事务。

@@TRANCOUNT 函数记录当前事务的嵌套级别。每个 BEGIN TRANSACTION 语句使@@TRANCOUNT 增加 1。每个 COMMIT TRANSACTION 语句使@@TRANCOUNT 减去 1。而 ROLLBACK TRANSACTION 语句将回滚所有嵌套事务，并使@@TRANCOUNT 减小到 0。在无法确定是否已经在事务中时，可以用 SELECT @@TRANCOUNT 确定@@TRANCOUNT 的值。如果@@TRANCOUNT 等于 0，则表明不在事务中。@@TRANCOUNT 函数最小是 0，不会出现负值。

例如创建一个事务 A，在 A 事务中再创建事务 B 和 C，然后提交事务 B，在 C 事务中回滚事务，通过@@TRANCOUNT 查看当前所在的事务嵌套级别。具体操作脚本如代码 21.5 所示。

代码 21.5　嵌套事务

```
SELECT @@TRANCOUNT            --输出 0
BEGIN TRAN A
    SELECT @@TRANCOUNT        --输出 1
    BEGIN TRAN B
        SELECT @@TRANCOUNT    --输出 2
    COMMIT TRAN B
    SELECT @@TRANCOUNT        --输出 1
    BEGIN TRAN C
        SELECT @@TRANCOUNT    --输出 2
    ROLLBACK
    SELECT @@TRANCOUNT        --输出 0
COMMIT TRAN A --这里会报错，因为 A 事务已经在前面回滚
```

21.1.4　事务保存点

在嵌套事务中一旦执行 ROLLBACK 命令就会将整个事务回滚，在实际应用中可能需要在事务中进行条件判断，然后回滚内部的事务。使用保存点可以标记出事务中哪个地方可以被回滚，从而实现了嵌套事务中的内部事务回滚。

SQL Server 中使用 SAVE TRANSACTION 语句来定义保存点，其语法格式为：

```
SAVE { TRAN | TRANSACTION } { savepoint_name | @savepoint_variable }
```

其中 savepoint_name 就是分配给保存点的名称。使用 SAVE TRAN 定义保存点并不影

响@@TRANCOUNT 的值，而 ROLLBACK 到保存点的动作也不会影响@@TRANCOUNT 的值。在事务中可以定义重复的保存点名称，但是在 ROLLBACK 时将回滚到使用该名称最近的一次 SAVE TRAN。

同样以 21.1.3 节中的嵌套事务示例为例，若使用事务保存点来回滚事务 C，则对应的 SQL 语句如代码 21.6 所示。

代码 21.6　嵌套事务中使用事务保存点

```
SELECT @@TRANCOUNT            --输出 0
BEGIN TRAN A
    SELECT @@TRANCOUNT        --输出 1
    BEGIN TRAN B
        SELECT @@TRANCOUNT    --输出 2
    COMMIT TRAN B
    SELECT @@TRANCOUNT        --输出 1
    SAVE TRAN C
        SELECT @@TRANCOUNT    --输出 1，因为 SAVE TRAN 不增加@@TRANCOUNT 值
    ROLLBACK TRAN C
    SELECT @@TRANCOUNT        --输出 1
COMMIT TRAN A
SELECT @@TRANCOUNT            --输出 0
```

保存点常用于使用了事务的存储过程中，在存储过程中如果有 ROLLBACK 语句，而调用该存储过程之前，若连接已经启用了事务，则 ROLLBACK 语句会回滚整个事务。一般是在存储过程中首先通过@@TRANCOUNT 判断当前是否在事务中，如果在事务中则执行 SAVE TRAN，如果没有在事务中则执行 BEGIN TRAN。

21.2　锁

锁定是 SQL Server 数据库引擎用来同步多个用户同时对同一个数据块访问的一种机制。在事务获取数据块当前状态的依赖关系（比如通过读取或修改数据）之前，事务使用锁来保护自己不受其他事务对同一数据进行修改的影响。本节将主要介绍锁的基础知识。

21.2.1　锁的模式

在事务发生时，SQL Server 数据库引擎会对事务相关的资源要求不同类型的锁定。SQL Server 数据库引擎使用不同的锁模式锁定资源，这些锁模式确定了并发事务访问资源的方式。在事务结束时系统会释放相关的锁以便其他事务使用。如表 21.1 列出了 SQL Server 中的资源锁模式。

表 21.1　锁模式

锁　模　式	说　　明
共享（S）	用于不更改或不更新数据的读取操作，如SELECT语句
更新（U）	用于可更新的资源中，是一种中间状态的锁，更新锁一般将升级为排他锁
排他（X）	用于数据修改操作，例如INSERT、UPDATE或DELETE
意向	用于建立锁的层次结构。意向锁包含3种类型：意向共享（IS）、意向排他（IX）和意向排他共享（SIX）

续表

锁　模　式	说　　明
架构	在执行依赖于表架构的操作时使用。架构锁包含两种类型：架构修改（Sch-M）和架构稳定性（Sch-S）
大容量更新（BU）	用于复制大量数据至表，并指定了TABLOCK提示时使用
键范围	当使用可序列化事务隔离级别时保护查询读取的行的范围。确保再次运行查询时其他事务无法插入符合可序列化事务查询的行

共享锁（Shared Lock，S 锁）是 SELECT 语句为对应资源添加的锁，允许并发事务在封闭式并发控制下读取资源。资源上存在共享锁时，任何其他事务都不能修改数据。在 SELECT 读取操作完成后就立即释放资源上的共享锁。若将事务隔离级别设置为可重复读或更高级别，或者在事务持续时间内用锁定提示保留共享锁，则事务中 SELECT 读取完后共享锁将不会释放。关于事务隔离级别将在 21.3 节进行讲解。

更新锁（Update Lock，U 锁）是在进行更新操作时为资源暂时添加的锁，可以防止常见的死锁。在默认的事务隔离级别可重复读或可序列化事务中，如果要执行一个查找更新（比如带 WHERE 条件的 UPDATE 语句），系统将先查找数据，在相关记录上放置共享锁。为了避免发生死锁，SQL Server 会对记录放置更新锁。由于更新锁与共享锁不相互排斥，所以两个事务都可以对同一资源放置共享锁，当一个事务尝试更新数据时，该事务便将共享锁提升到更新锁，而更新锁之间相互排斥，所以另一个事务就将进入等待状态。

前面说到，一般情况下共享锁在 SELECT 语句结束后就会立即释放，在共享锁释放后更新锁就可以上升到排他锁，而后更新数据。所以更新锁是在查找更新数据时，从共享锁上升到排他锁之间的一个过渡锁定。

排他锁（Exclusive Lock，X 锁）又叫独占锁，可以防止并发事务对资源进行访问。使用排他锁时，任何其他事务都无法修改数据，仅在使用 NOLOCK 提示或未提交读隔离级别时才会进行读取操作。排他锁并不像共享锁一样在语句执行完成后立即释放，而是会在事务开始后一直保留到事务结束为止。

数据修改语句（如 INSERT、UPDATE 和 DELETE）合并了修改和读取操作。语句在执行所需的修改操作之前首先执行读取操作以获取数据。因此，数据修改语句通常请求共享锁和排他锁。

意向锁（Intent Lock）用于保护共享锁或排他锁放置在锁层次结构的底层资源上。之所以命名为意向锁，是因为在较低级别锁前可获取它们，因此会通知意向将锁放置在较低级别上。意向锁有两种用途：

- ❑ 防止其他事务以会使用较低级别的锁以无效的方式修改较高级别资源。
- ❑ 提高数据库引擎在较高的粒度级别检测锁冲突的效率。

意向锁包括意向共享（IS）、意向排他（IX）以及意向排他共享（SIX）等，具体参见表 21.2。

表 21.2　意向锁模式

锁　模　式	说　　明
意向共享（IS）	保护针对层次结构中某些低层资源请求或获取的共享锁
意向排他（IX）	保护针对层次结构中某些低层资源请求或获取的排他锁。IX是IS的超集，它也保护针对低层级别资源请求的共享锁
意向排他共享（SIX）	保护针对层次结构中某些低层资源请求或获取的共享锁，以及针对某些低层资源请求或获取的意向排他锁。顶级资源允许使用并发IS锁

续表

锁　模　式	说　　明
意向更新（IU）	保护针对层次结构中所有低层资源请求或获取的更新锁。仅在页资源上使用IU锁。如果进行了更新操作，IU锁将转换为IX锁
共享意向更新（SIU）	S锁和IU锁的组合，作为分别获取这些锁并且同时持有两种锁的结果
更新意向排他（UIX）	U锁和IX锁的组合，作为分别获取这些锁并且同时持有两种锁的结果

架构锁，数据库引擎在表数据定义语言操作（例如添加列或删除表）的过程中，将对操作相关数据库对象使用架构修改（Sch-M）锁。架构修改锁与独占锁类似，在释放前将阻止所有外围操作。

数据库引擎在编译和执行查询时使用架构稳定性（Sch-S）锁。架构稳定锁不会阻止某些事务锁，其中包括排他锁。因此，在编译查询的过程中其他使用排他锁的事务将继续运行。

大容量更新锁，数据库引擎在将数据大容量复制到表中时使用了大容量更新锁（Bulk Update，BU 锁），并指定了 TABLOCK 提示或使用 sp_tableoption 设置了 table lock on bulk load 表选项。大容量更新锁一方面允许多个线程将数据并发地大容量加载到同一表，另一方面要防止其他不进行大容量加载数据的进程访问该表，以提高大容量操作的性能。

键范围锁，在使用可序列化事务隔离级别时，对于 T-SQL 语句读取的记录集，键范围锁可以隐式保护该记录集中包含的行范围。键范围锁可防止幻读（21.3 节将介绍什么是幻读）。通过保护行之间键的范围，它还防止对事务访问的记录集进行幻象插入或删除。

使用系统存储过程 sp_lock 可以查看指定会话或者整个数据库中的锁信息。存储过程 sp_lock 支持两个参数，都是要查看锁信息的会话 ID，如果传入一个会话 ID 则返回该会话 ID 当前所占用的锁。如果传入 2 个会话 ID，则返回 2 个会话所占用的锁。如果不传入任何参数，则返回整个数据库的锁情况。例如要查询 53 号会话当前的锁信息，则查询脚本为：

```
EXEC sp_lock 53
```

注意：在 SQL Server 2005 之后还可以通过查询动态管理视图 sys.dm_tran_locks 获得当前数据库的锁信息。建议使用动态管理视图，因为 sp_lock 作为过时的存储过程，将会在以后的版本中删除。

21.2.2　锁的兼容性

锁兼容性用于在多个事务中决定是否能够同时获取同一资源上的锁。如果资源已被另一事务锁定，则仅当请求锁的模式与现有锁的模式相兼容时，才会授予新的锁请求。如果请求锁的模式与现有锁的模式不兼容，则请求新锁的事务将处于等待状态，等占有该资源的事务释放现有锁或等待锁超时过期。

例如，排他锁是与其他锁不兼容的。如果事务 A 对表 t1 具有排他锁，则事务 B 无论是查询还是更新表 t1，由于表 t1 上的排他锁与其他锁不兼容，所以 B 事务将处于等待状态。在 A 事务释放排他锁之前，其他事务均无法获取该资源任何类型的锁。

对于兼容的锁模式，多个事务都可以使用该资源。例如共享锁可以与共享锁兼容。A 事务在表 t1 上请求了共享锁，此时 B 事务再去对表 t1 请求共享锁则不需要等待 A 事务释放它的锁。如表 21.3 列出了常见的锁模式之间的兼容性。

表 21.3　常见锁模式兼容性

	当前已授予的模式					
请求模式	IS	S	U	IX	SIX	X
意向共享（IS）	是	是	是	是	是	否
共享（S）	是	是	是	否	否	否
更新（U）	是	是	否	否	否	否
意向排他（IX）	是	否	否	是	否	否
意向排他共享（SIX）	是	否	否	否	否	否
排他（X）	否	否	否	否	否	否

注意：虽然排他锁与其他锁不兼容，但是意向排他锁却可以与意向共享和意向排他锁兼容，因为 IX 表示打算只更新部分行而不是所有行。还允许其他事务尝试读取或更新部分行，只要这些行不是其他事务当前更新的行即可。

21.2.3　锁的资源和粒度

SQL Server 数据库引擎提供了多种数据粒度锁定，根据用户访问对象行为模式的不同，允许一个事务锁定不同类型的资源。

为了尽量减少锁定的开销，数据库引擎自动将资源锁定在适合任务的级别。较小粒度的锁定可以减小其他事务访问资源进行等待的几率，从而提高并发度，但开销较高。如果锁定的粒度较低，则需要持有更多的锁。锁定在较大的粒度（例如锁定整个表）上会限制其他事务对表中任意部分的访问，从而降低并发度，而且其开销较低，需要维护的锁较少。

SQL Server 数据库引擎在锁定资源时，通常是从粒度较小的对象开始，避免大范围地锁定导致其他事务的阻塞。但是由于粒度太小会造成所使用的锁较多，要求的系统资源更多，在系统资源不足的情况下 SQL Server 会自动扩大锁定范围从而降低系统资源占用。

SQL Server 必须获取多粒度级别上的锁才能完整地保护资源，这组多粒度级别上的锁称为锁层次结构。例如，为了完整地保护对索引的读取，数据库引擎实例可能必须获取行上的共享锁及页和表上的意向共享锁。如表 21.4 列出了数据库引擎可以锁定的资源。

表 21.4　锁定的资源

资　　源	缩　　写	说　　明
RID	RID	用于锁定堆中单个行的行标识符
KEY	KEY	索引中用于保护可序列化事务中的键范围的行锁
PAGE	PAG	数据库中的8KB页，例如数据页或索引页
EXTENT	EXT	一组连续的8页，例如数据页或索引页
HoBT	IDX	堆或B树。用于保护没有聚集索引的表中的B树（索引）或堆数据页的锁
TABLE	TAB	包括所有数据和索引的整个表
FILE	FIL	数据库文件
APPLICATION	APP	应用程序专用的资源
METADATA	MD	元数据锁
ALLOCATION_UNIT	AU	分配单元
DATABASE	DB	整个数据库

21.3　事务隔离级别

SQL Server 为事务提供了不同的隔离级别，用于指定事务中对资源的锁定情况或数据更改时事务之间相隔离的程度。不同的隔离级别将针对不同并发进行控制。

21.3.1　并发产生的影响

一个用户对表中的数据进行修改，同时将会影响到其他用户对该表的读取或修改相同的数据。在并发情况下，这些用户都可以同时访问数据。如果数据存储系统没有并发控制，则用户可能会看到以下负面影响：

- 丢失更新；
- 未提交的依赖关系（脏读）；
- 不一致的分析（不可重复读）；
- 幻读；
- 由于行更新导致读取缺失和重复读。

丢失更新是指当两个事务 A 和 B 同时更新同一行数据，由于每个事务都不知道其他事务的存在，A 事务更新了数据后 B 事务又进行了更新，所以 B 做的最后更新将覆盖由 A 事务所做的更新，这将导致数据丢失。在 A 事务看来，其更新丢失了。

例如，如果两个程序员都在修改同一个源代码文件，他们都将源代码进行了各自的修改后再覆盖服务器上的原始代码，那么先保存的代码就会被后保存的代码给覆盖。如果一个程序员在修改代码时锁定该代码，不允许其他程序员修改，只有等代码签入到服务器上后才允许其他用户签出进行修改，则可以避免这种问题。实际上 VSS（微软的一款源代码管理工具）就可以通过对代码的锁定来保证不会造成丢失更新的代码。

未提交的依赖关系（脏读）是指当事务 A 正在对一个数据进行更新，更新后事务 A 还没有提交，同时事务 B 要对这些数据进行读取，这时事务 B 会将事务 A 未提交的更新读取出来。由于事务 A 还没有提交，所以其一旦回滚，事务 B 中读取出来的数据就包含脏数据，所以叫做脏读。

同样以前面提到的两个开发人员为例，一个开发人员 A 正在对代码进行修改，修改了一部分功能的时候，另一个开发人员 B 从 A 那里复制了修改后的代码，然后编译成程序。此后，A 觉得对代码的修改有问题，所以撤销了对代码的修改（在 VSS 中对应的功能就是撤销本地更改）。此时从源代码服务器上获得的源代码与 B 手里的源代码是不一样的，如果在一个程序员没有提交其代码的修改之前，任何人都不可以去复制其修改后的代码，则可以避免这种情况发生。

不一致的分析（不可重复读）是指 A 事务对同一行数据进行了多次的读取，但是在两次读取之间，B 事务对数据进行了更改并提交更改，此时 A 事务先读取的数据与后读取的数据不相同，因此我们称之为“不可重复读”。

例如，开发人员 A 要获得源代码的多个副本，在获取了一个文件后，另一个开发人员

B 对该文件进行了签出修改并提交，此时 A 第二次获取该文件时获得的就是 B 修改后的文件，与第一次获取的文件不相同，也就是说这两次读取不可重复。解决办法是在一个用户正在获取文件时其他用户不可以对文件进行修改，这样便可以避免此问题。

幻读是指一个事务 A 要读取一个范围内的数据，第一次读取了该范围的数据后事务 A 还未提交，此时事务 B 对该范围内的数据执行了插入或删除操作，事务 A 进行第二次读取将出现相对于第一次读取多了数据或者少了数据的情况。

例如，一个开发人员 A 更改代码，删除了一个代码文件，但是在 A 提交删除文件之前，开发人员 B 正在读取整个源代码，在提交删除之后开发人员 B 再次获取整个源代码时将发现与上一次获取的代码文件数量不相同。与不可重复读的情况类似，如果在开发人员 B 获取整个源代码的过程中其他任何人都不允许添加或者删除文件，则可以避免该问题。

21.3.2　隔离级别概述

事务隔离级别定义一个事务必须与其他事务所进行的资源或数据更改相隔离的程度。隔离级别从允许的并发副作用（例如，脏读或幻读）的角度进行描述。

事务隔离级别主要控制以下内容：

- 读取数据时是否占用锁及所请求的锁类型。
- 占用读取锁的时间。
- 引用其他事务修改的行的读取操作，是否在该行上的排他锁被释放之前阻塞其他事务，检索在启动语句或事务时存在行的已提交版本，读取未提交的数据修改。

事务隔离级别并不影响为保护数据修改而获取的锁。也就是说，不管设置的是什么事务隔离级别，事务总是在其修改的任何数据上获取排他锁，并在事务完成之前都会持有该锁。在读取操作中，不同的事务隔离级别主要定义不同的保护粒度和保护级别，以防受到其他事务所做更改的影响。

事务隔离级别低，则保护级别低，保护粒度小，所以可以增强并发访问数据的能力，但同时也增加了用户访问数据时可能遇到的并发副作用的可能性。相反，较高的隔离级别减少了用户可能遇到的并发副作用的类型，但需要更多的系统资源，并增加了一个事务阻塞其他事务的可能性。在实际应用中要根据实际业务需要平衡数据完整性与每个隔离级别的开销，选择合适的事务隔离级别。默认情况下，系统使用已提交读的事务隔离级别。

除了默认的已提交读外，ISO 标准还定义了下列隔离级别，SQL Server 数据库引擎支持所有这些隔离级别。

- 未提交读（隔离事务的最低级别，只能保证不读取物理上损坏的数据）；
- 已提交读（数据库引擎的默认级别）；
- 可重复读；
- 可序列化（隔离事务的最高级别，事务之间完全隔离）。

除了以上事务隔离级别外，SQL Server 2005 还增加了支持使用行版本控制的两个事务隔离级别。一个是已提交读隔离的新实现，另一个是新事务隔离级别（快照）。如表 21.5 列出了隔离级别导致的并发副作用。

表 21.5　隔离级别副作用

隔 离 级 别	脏　　读	不可重复读	幻　　读
未提交读	是	是	是
已提交读	否	是	是
可重复读	否	否	是
快照	否	否	否
可序列化	否	否	否

从表 21.5 中可以看到，未提交读参数的副作用最大，而可序列化隔离级别将不会产生这几种副作用。

21.3.3　使用 T-SQL 设置隔离级别

T-SQL 中可以通过 SET TRANSACTION ISOLATION LEVEL 语句，设置当前会话的事务隔离级别，其语法如代码 21.7 所示。

代码 21.7　设置隔离级别

```
SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SNAPSHOT
    | SERIALIZABLE
    }
```

例如将事务隔离级别设置为 READ COMMITTED，然后在事务中读取表中的数据，对应的 SQL 脚本如代码 21.8 所示。

代码 21.8　设置隔离级别并进行事务操作

```
USE AdventureWorks2012;
GO
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;   --设置为可重复读
GO
BEGIN TRANSACTION;  --开始事务
GO
SELECT *
FROM HumanResources.Employee;
GO
SELECT *
FROM HumanResources.Department;
GO
COMMIT TRANSACTION; --提交事务
```

事务隔离级别的设置是与连接相关的，一个连接中只能同时存在一个隔离级别，而且设置的选项将对那个连接始终有效，直到显式更改该选项为止。事务中执行的所有读取操作都会在指定的隔离级别规则下运行，除非语句的 FROM 子句中的表提示为表指定了其他锁定行为或版本控制行为。

在事务进行期间，可以随时将事务从一个隔离级别切换到另一个隔离级别，但从任一

隔离级别更改到 SNAPSHOT 隔离时例外。不过可以将 SNAPSHOT 隔离中启动的事务更改为任何其他隔离级别。

如果将事务从一个隔离级别更改为另一个隔离级别，那么事务在之后便会根据新级别的规则对更改后读取的资源执行保护，而在更改前读取的资源，将继续按照以前级别的规则受到保护。例如，某个事务先前的隔离级别是已提交读，现在更改为可序列化，则在该事务结束前，由于可序列化级别的共享锁将保留，所以更改后所获取的共享锁将一直处于保留状态。

事务隔离级别在存储过程被调用时将保存当前的隔离级别，也就是说，如果在存储过程内部使用了 SET TRANSACTION ISOLATION LEVEL，则在调用存储过程之前，系统保存当前事务隔离级别，等调用的存储过程返回控制时，隔离级别会重设为在调用该存储过程之前有效的级别。

例如代码 21.9 所示，创建一个存储过程 spA，在该存储过程中将事务隔离级别设置为可序列化，而外部再开启了一个事务，当前的事务隔离级别是可重复读，而在该事务中调用了存储过程 spA，那么存储过程内部的事务就是可序列化的，而调用存储过程结束后，当前的事务又回到可重复读的状态。

代码 21.9　存储过程中设置事务隔离级别

```
USE AdventureWorks2012;
GO
CREATE PROC spA                                 --创建存储过程
AS
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE    --设置为可序列化隔离级别
SELECT *
FROM Person.AddressType
GO
--设置事务隔离级别为可重复读，然后启用事务
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
SELECT TOP 10 *          --这里的事务隔离级别是可重复读
FROM Person.Address
EXEC spA                 --调用存储过程，存储过程内部的事务隔离级别是可序列化
SELECT *
FROM Person.ContactType  --这里的事务隔离级别是可重复读
COMMIT
```

注意：SET TRANSACTION ISOLATION LEVEL 会在执行或运行时生效，而不是在分析时生效。

21.3.4　隔离级别详情

使用 SET TRANSACTION ISOLATION LEVEL 命令可以设置不同的事务隔离级别，在对隔离级别有了基本的概念后接下来就分别介绍每个隔离级别的特点。

1．未提交读

指定语句可以读取已由其他事务修改但尚未提交的行。

设置在 READ UNCOMMITTED 级别运行的事务，可以读取其他事务未提交的修改，当前事务也不会发出共享锁来防止其他事务修改当前事务读取的数据，也就是脏读。另外，在 READ UNCOMMITTED 隔离级别下，事务也不会被排他锁阻塞，前面已经说到排他锁和共享锁是不兼容的，但是由于在 READ UNCOMMITTED 隔离级别下读取数据并不会请求共享锁，所以也就不会存在排他锁对当前事务的阻塞。

例如在 SSMS 中新建一个查询窗口（也就是新建了一个数据库连接），在该查询窗口中启用事务对 Person.AddressType 进行修改，而该事务并没有提交，具体脚本如代码 21.10 所示。

代码 21.10　启用事务修改表

```
USE AdventureWorks2012;
GO
BEGIN TRAN A --启用了事务 A，但是该事务并没有提交
UPDATE Person.AddressType
SET ModifiedDate=GETDATE()
```

接下来再新建一个查询窗口，在该窗口中设置事务隔离级别为未提交读，然后查询 AddressType 表，系统返回了表的内容，ModifiedDate 字段已经被修改为当前日期，也就是说事务 A 未提交的修改已经在未提交读隔离级别下被查询出来，具体脚本如代码 21.11 所示。

代码 21.11　使用未提交读事务隔离级别查询数据

```
USE AdventureWorks2012;
GO
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SELECT *
FROM Person.AddressType --该查询返回了事务 A 修改后的数据
```

在未提交读隔离级别下已经出现了脏读，在第一个查询窗口中运行 ROLLBACK 命令将事务 A 回滚，则代码 21.11 返回的结果又变成了未修改前的状态。未提交读事务隔离级别的作用，与在事务内所有 SELECT 语句中的所有表上设置 NOLOCK 相同。例如再新建一个查询窗口，使用默认的事务隔离级别，则使用 NOLOCK 方式读取事务 A 对 AddressType 表的修改的脚本如代码 21.12 所示。

代码 21.12　使用 NOLOCK 方式读取未提交的更改

```
USE AdventureWorks2012;
GO
SELECT *
FROM Person.AddressType WITH(NOLOCK) -读取了事务 A 未提交的更改
```

2. 已提交读

已提交读使用 READ COMMITTED 选项，是 SQL Server 的默认隔离级别。在已提交读隔离级别下的事务不能读取已由其他事务修改但尚未提交的数据，这样可以避免脏读。但是其他事务可以在当前事务的各个语句之间更改数据，从而产生不可重复读取和幻读。

在已提交读隔离级别下，数据库引擎会使用共享锁防止其他事务在当前事务执行读取

操作期间修改行。共享锁还会阻止语句在其他事务完成之前读取由这些事务修改的行。共享锁并不是在事务提交的时候释放的，而且在进行下一个处理之前释放：行锁在处理下一行之前释放、页锁在读取下一页时释放、表锁在语句完成时释放。

已提交读可以防止脏读，以前面未提交读使用的示例为例，将第二个查询窗口的隔离级别设置为 READ COMMITTED，则该查询将会被一直阻塞，不会查询出任何结果，直到事务 A 提交或者回滚。

由于在已提交读隔离级别下共享锁并不是一直保持到事务结束，所以在两次读取之间并没有对相关对象放置共享锁，这将造成不可重复读。例如新建一个查询窗口，将其设置为已提交读隔离级别，启用事务读取 AddressType 表，该事务并没有提交，具体脚本如代码 21.13 所示。

代码 21.13　在已提交读隔离级别读取数据

```
USE AdventureWorks2012;
GO
SET TRANSACTION ISOLATION LEVEL READ COMMITTED --默认的已提交读
BEGIN TRAN A
SELECT *
FROM Person.AddressType
```

接下来新建一个查询窗口，在该窗口中对 AddressType 进行更新，具体脚本如代码 21.14 所示。

代码 21.14　更新 AddressType 表的内容

```
USE AdventureWorks2012;
GO
UPDATE Person.AddressType
SET ModifiedDate=GETDATE()
```

更新成功后接下来再回到事务 A 所在的窗口，再次运行查询脚本：

```
SELECT *
FROM Person.AddressType
```

这时可以看到系统返回的数据已经是被修改后的数据，这就是不可重复读现象。

3. 可重复读

可重复读使用 REPEATABLE READ 选项，指定语句不能读取已由其他事务修改但尚未提交的行，同时其他任何事务都不能在当前事务完成之前，修改由当前事务读取的数据。这样可以避免不可重复读现象。

可重复读之所以能够防止其他事务修改当前事务读取过的数据，是因为在该隔离级别下，系统对事务中的每个语句所读取的全部数据都设置了共享锁，并且该共享锁一直保持到事务完成为止。由于共享锁与排他锁不兼容，所以可以防止其他事务修改当前事务读取的任何行。例如对于已提交读中的示例，将代码 21.13 中的事务隔离级别设置为 REPEATABLE READ，则代码 21.14 中对表的更新将会被阻塞，直到事务 A 完成为止。

虽然可重复读隔离级别下可以防止其他事务更新对象，但是并不能防止其他事务插入与当前事务所发出语句的搜索条件相匹配的新行。在插入新行后，当前事务随后重试执行

查询语句将会检索到插入的新行，从而产生幻读。例如新建一个查询窗口，在其中使用可重复读隔离级别开启一个事务 A，在事务中读取 AddressType 中的数据，该事务并没有提交，具体脚本如代码 21.15 所示。

代码 21.15　在可重复读隔离级别读取数据

```
USE AdventureWorks2012;
GO
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ    --设置为可重复读
BEGIN TRAN A
SELECT *
FROM Person.AddressType
```

可以看到系统返回了 6 行结果，然后新建一个查询窗口，在其中向 AddressType 表插入一行数据，具体脚本如代码 21.16 所示。

代码 21.16　向 AddressType 插入数据

```
USE AdventureWorks2012;
GO
INSERT INTO Person.AddressType
VALUES('Test',NEWID(),GETDATE())
```

代码运行成功后再回到事务 A 所在的查询窗口，再次查询 AddressType 表，查询脚本仍然是：

```
SELECT *
FROM Person.AddressType
```

系统返回了 7 行结果，而先前返回的是 6 行结果，这就是幻读现象。

注意：由于可重复读隔离级别下共享锁一直保持到事务结束，而不是在每个语句结束时释放，所以并发级别低于默认的 READ COMMITTED 隔离级别。此选项只在必要时使用。

4．快照

快照隔离级别是 SQL Server 2005 中增加的一种隔离级别，在 SQL Server 2012 中也可以使用。默认情况下数据库并不允许使用快照隔离级别，必须使用 ALTER DATABASE 命令将 ALLOW_SNAPSHOT_ISOLATION 数据库选项设置为 ON，才能开始一个使用 SNAPSHOT 隔离级别的事务。

在快照隔离级别下的事务中，任何语句读取的数据都将是在事务开始时便存在的。当前事务只能识别在其开始之前提交的数据修改，在当前事务中执行的语句将看不到在其开始以后由其他事务所做的数据修改。

在前面的章节中介绍过数据库快照技术，这里的快照隔离级别相当于对数据进行了快照，快照隔离级别下的事务不会在读取数据时请求锁，所以也就不会阻止其他事务写入数据。其他事务对源表进行修改，而当前事务读取的数据来自于快照，所以总是不变的。例如使用快照隔离级别开启事务 A，对 AddressType 进行读取，事务并没有提交，具体脚本如代码 21.17 所示。

代码 21.17　使用快照隔离级别读取数据

```
ALTER DATABASE AdventureWorks2012 SET ALLOW_SNAPSHOT_ISOLATION ON
--使数据库允许快照隔离级别
USE AdventureWorks2012;
GO
SET TRANSACTION ISOLATION LEVEL SNAPSHOT   --使用快照事务隔离级别
BEGIN TRAN A
SELECT *
FROM Person.AddressType
```

然后新建一个查询窗口，在该查询窗口中对 AddressType 进行更新，具体脚本如代码 21.18 所示。

代码 21.18　更新 AddressType 表

```
USE AdventureWorks2012;
GO
UPDATE Person.AddressType       --更新 AddressType 表
SET ModifiedDate=GETDATE()
GO
INSERT INTO Person.AddressType --插入新数据
VALUES('Test1',NEWID(),GETDATE())
```

并不像可重复读隔离级别那样，系统并没有阻塞对表的更新，更新成功，向表 AddressType 中插入一条数据也成功。再回到事务 A，重新对表 AddressType 进行查询，查询脚本为：

```
SELECT *
FROM Person.AddressType
```

系统返回的结果和第一次查询的结果相同，也就是说没有发生不可重复读和幻读现象。快照隔离级别使用了与其他隔离级别不同的方案，解决了并发情况下产生的脏读、不可重复读和幻读现象。

注意：SNAPSHOT 隔离级别在每个数据库中默认都是关闭的，如果使用 SNAPSHOT 隔离级别的事务访问多个数据库中的数据，则必须在每个数据库中将 ALLOW_SNAPSHOT_ISOLATION 都设置为 ON。

5. 可序列化

可序列化隔离级别使用 SERIALIZABLE 选项，是最高级别的事务隔离级别，实现了事务之间的完全隔离。在可序列化隔离级别下的语句，不能读取已由其他事务修改但尚未提交的数据，同时任何其他事务都不能在当前事务完成之前修改由当前事务读取的数据，也不能使用当前事务中任何语句读取的键值插入新行，从而防止了脏读、不可重复读和幻读现象。

可序列化隔离级别使用了范围锁来阻止其他事务更新或插入任何行，在事务完成之前将一直保持范围锁。这是限制最多的隔离级别，所以应只在必要时才使用该选项。例如使用可序列化隔离级别开启事务 A，在 A 事务中查询 AddressType 表，并且该事务没有提交，具体脚本如代码 21.19 所示。

代码 21.19　使用可序列化隔离级别查询数据

```
USE AdventureWorks2012;
GO
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE   --设置为可序列化
BEGIN TRAN A
SELECT *
FROM Person.AddressType
```

然后新建查询窗口，向 AddressType 表中插入一行新数据，则对应的脚本如代码 21.20 所示。

代码 21.20　向 AddressType 中插入数据

```
USE AdventureWorks2012;
GO
INSERT INTO Person.AddressType     --插入数据
VALUES('Test3',NEWID(),GETDATE())
```

运行代码 21.20 将会被系统阻塞，直到事务 A 完成为止，从而保证了不会发生幻读。可序列化选项的作用，与在事务内所有 SELECT 语句中的所有表上设置 HOLDLOCK 相同，例如运行代码 21.21 查询数据，然后再运行 21.20 同样会阻塞对表的插入。

代码 21.21　使用 HOLDLOCK 锁定表

```
USE AdventureWorks2012;
GO
BEGIN TRAN A
SELECT *
FROM Person.AddressType WITH(HOLDLOCK) --锁定表选项
```

21.4　死　　锁

在 SQL Server 事务处理中，如果多个任务的资源访问相互锁定将会发生死锁，造成死锁中的一个事务被回滚终止。本节主要讲解死锁的相关知识。

21.4.1　死锁简介

在两个或多个任务中，如果每个任务锁定都锁定了一定资源，然后试图锁定其他事务已经锁定的资源，此时会造成这些任务相互等待对方释放资源，形成永久阻塞，从而出现死锁。例如：

（1）事务 1 获取了表 t1 的排他锁。

（2）事务 2 获取了表 t2 的排他锁。

（3）现在，事务 1 请求表 t2 的更新锁，由于更新锁与排他锁不兼容，所以事务 1 必须等待事务 2 释放对 t2 的锁定。

（4）接下来，事务 2 请求表 t1 的更新锁，同样，事务 2 将会被事务 1 阻塞，等待事务 1 释放对表 t1 的锁定。

现在事务 A 在等待事务 B 释放资源，事务 B 也在等待事务 A 释放资源，从而形成了

死循环，如图 21.1 所示。

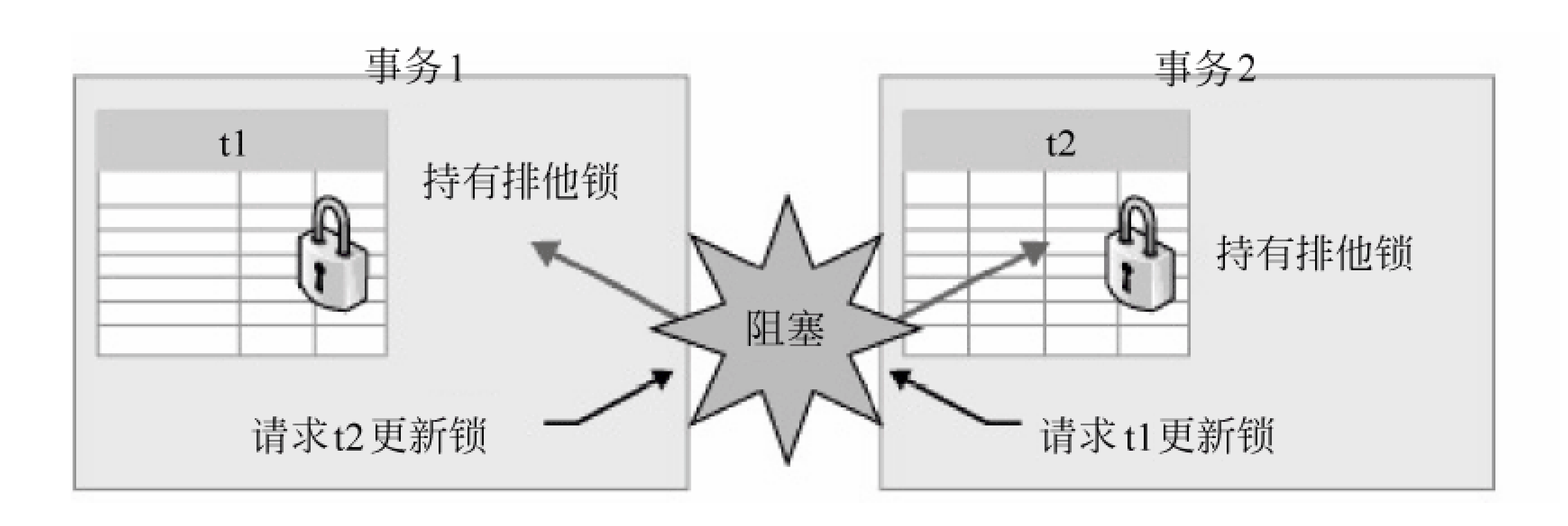


图 21.1　死锁图

死锁形成后除非被某个外部进程断开，否则死锁中的两个事务都将无限期等待下去，而资源也会被无限期地锁定。SQL Server 的数据库引擎内部有专门的死锁监视器定期（默认时间间隔为 5 秒）检查陷入死锁的任务。监视器检测死锁，将选择其中一个事务作为牺牲品，然后回滚事务操作，终止事务并提示 1205 错误，另一个事务则可以顺利完成。

死锁与正常阻塞不同，死锁是形成了循环依赖关系，而正常阻塞只是一个事务正在运行，占用了资源而阻塞了其他希望访问该资源的事务。默认情况下，除非设置了 LOCK_TIMEOUT，否则 SQL Server 事务不会超时，如果一个事务一直没有提交，那么将会一直阻塞其他相关事务，但是只要不形成循环依赖，则不能认为是死锁。

死锁现象不仅发生在关系数据库管理系统中，任何多线程系统上都会发生死锁。做过多线程编程的人员应该知道，在修改一个对象（比如一个文本框、一个按钮）时都要请求锁定该对象，如果该对象已经被其他线程锁定，则只有等待其他线程释放资源。如果两个线程相互锁定了对方请求的对象，则会发生多线程死锁。

21.4.2　多表死锁

通常情况下，死锁最容易发生在两个表之间。例如现在有两个会话（在 SSMS 中就是两个查询窗口），会话 A 中启用了事务 t1，执行了对 Person.AddressType 表的 UPDATE 操作，具体脚本如代码 21.22 所示。

代码 21.22　启用事务 t1 执行修改操作

```
BEGIN TRAN t1   --开启一个事务更新表
UPDATE Person.AddressType
SET ModifiedDate=GETDATE()
```

运行了以上代码后接下来在会话 B 中启用事务 t2，在该事务中对 Person.AddressType 表进行 UPDATE 操作，具体操作脚本如代码 21.23 所示。

代码 21.23　启用事务 t2 执行修改操作

```
BEGIN TRAN t2   --开启另一个事务更新表
UPDATE Person.Address
SET ModifiedDate=GETDATE()
```

现在 t1 事务已经对 AddressType 添加了排他锁，t2 事务对 Address 表添加了排它锁，

接下来再回到会话 A，在 t1 事务中更新 Address 表，具体代码如下所示。

```
UPDATE Person.Address   --更新其他表
SET ModifiedDate=GETDATE()
```

由于 Address 表已经被 t2 添加了排他锁，而且 t2 事务没有提交或回滚，所以 t1 对 Address 表的访问将被阻塞。

切换到会话 B 中，在事务 t2 中执行对 AddressType 的更新修改，具体代码如下所示。

```
UPDATE Person.AddressType         --交叉更新表
SET ModifiedDate=GETDATE()
```

由于 AddressType 被 t1 事务添加了排他锁，所以 t2 对 AddressType 表的修改将会被 t1 阻塞。现在 t1 阻塞了 t2，t2 也阻塞了 t1，死锁就此形成。

几秒钟后，会话 A 中将会收到一条异常消息：

```
消息 1205，级别 13，状态 56，第 1 行
事务(进程 ID 52)与另一个进程被死锁在 锁 资源上，并且已被选作死锁牺牲品。请重新运行该事务。
```

这就是死锁监视器检测到死锁后，将其中的一个事务作为牺牲品进行了回滚。而切换到会话 B 中将看到事务 t2 对 AddressType 的修改已经完成。这里系统选择会话 A 中的事务作为牺牲品，如果希望指定在发生死锁时应该牺牲哪个事务，可以通过使用 SET DEADLOCK_PRIORITY 命令来设置会话的优先级。其语法格式为：

```
SET DEADLOCK_PRIORITY { LOW | NORMAL | HIGH | <numeric-priority> | -10 |
-9 | -8 | … | 0 | … | 8 | 9 | 10 }
```

说明： 这里只是一个死锁试验，为了防止正在对数据库进行修改，在 B 会话中执行 ROLLBACK 命令回滚 t2 事务。

21.4.3　高隔离级别造成单表死锁

死锁不一定是发生在两个表之间，多个事务对单个表中的资源相互锁定也会造成死锁。例如在可重复读隔离级别下，事务 A 先查询 AddressType 表，对表放置共享锁，具体 SQL 脚本如代码 21.24 所示。

代码 21.24　创建测试用表和数据

```
USE AdventureWorks2012;
GO
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ    --设置为可重复读
BEGIN TRAN A                                       --启用事务读取数据
SELECT *
FROM Person.AddressType
```

事务 B 也查询 AddressType 表，对表放置共享锁，代码类似。由于在可重复读隔离级别下，所以这些共享锁是不释放的。接下来事务 A 中对表 AddressType 进行更新操作，具体代码如下所示。

```
UPDATE Person.AddressType  --更新表中的数据
SET ModifiedDate=GETDATE()
```

由于 AddressType 上被事务 B 放置了共享锁，所以事务 A 对表的更新操作将会被事务 B 阻塞。接下来事务 B 中也执行同样的更新操作，由于事务 A 对表持有共享锁，所以 B 会被 A 事务阻塞。现在死锁已经形成，其中一个事务将作为牺牲品被回滚。

21.4.4　索引建立不当造成单表死锁

前面介绍的单表死锁的发生是由于事务隔离级别的等级太高，造成了事务一直对表占有共享锁，从而产生死锁，只需要降低事务隔离级别，设置隔离级别为未提交读或者已提交读即可。但并不是使用了较低的事务隔离级别就不会发生单表死锁，事务之间争夺的对象不仅仅是表，还有索引，如果索引建立不当，事务之间争夺表的索引资源也会造成死锁。

例如在测试数据库中创建一个测试表 t1，该表上有聚集索引（建立主键时默认创建）和一个非聚集索引，然后向该表中添加一些数据，具体脚本如代码 21.25 所示。

代码 21.25　创建单表死锁测试表

```
CREATE TABLE t1
(
c1 INT NOT NULL PRIMARY KEY,        --聚集索引
c2 INT NOT NULL,
c3 VARCHAR(8)
)
GO
CREATE INDEX IX t1 c2 ON t1(c2)     --非聚集索引
SET NOCOUNT ON
DECLARE @i INT=1                    --添加 1000 条测试数据
WHILE @i<=1000
BEGIN
    INSERT INTO t1 VALUES(@i,@i,'test')
    SET @i+=1
END
```

创建好测试表后，接下来新建一个查询窗口 A，在该窗口中写一个死循环，不断地对 t1 表执行 UPDATE 操作，将建有非聚集索引的 c2 字段的值在 100～9900 之间轮换，具体脚本如代码 21.26 所示。

代码 21.26　对表执行 UPDATE 操作

```
SET NOCOUNT ON       --不显示受影响的行数
WHILE 1=1            --死循环
BEGIN
    UPDATE t1
    SET c2=10000-c2 --在 100～9900 之间轮换
    WHERE c1=100
END
```

接下来再创建一个查询窗口 B，在其中对 t1 表进行查询，返回 c3 字段，该字段所在行的 c2 字段值为 100，具体脚本如代码 21.27 所示。

代码 21.27　对表执行 SELECT 操作

```
WHILE 1=1
BEGIN
    SELECT c3   --查询数据
    FROM t1
    WHERE c2=100
END
```

在运行一段时间后系统中将发生死锁，查询窗口 B 将抛出 1205 错误。这个死锁是发生在表 t1 的聚集索引和非聚集索引上，下面就来分析一下死锁是怎么发生的。

查询 A 对 c2 字段进行更新，总共进行了 2 步操作，先是通过 WHERE 条件找到该行数据，然后再对 c2 字段进行更新。在 WHERE 条件中使用了 c1 进行查找，系统将在 c1 所在的聚集索引上放置排他锁。由于 c2 字段上建立了非聚集索引，所以系统在更新 c2 字段的数据时还要更新 c2 所在的非聚集索引，因此系统将请求对非聚集索引的排他锁。

查询 B 是一个带 WHERE 条件的查询，总共也是进行了 2 步操作。由于是使用 c2 作为查询条件，所以先是使用 c2 字段上的非聚集索引找到数据行，在非聚集索引上放置共享锁。接下来要查找 c3 字段，由于 c3 字段没有在非聚集索引上，所以需要通过聚集索引查找到符合条件的行，然后找到 c3 字段，因此需要请求聚集索引上的共享锁。

查询 A 想获得非聚集索引上的排他锁以更新 c2 数据，但是非聚集索引已经被查询 B 放置了共享锁。查询 B 想获得聚集索引上的共享锁以获得 c3 数据，但是聚集索引已经被查询 A 放置了排他锁。就这样死锁便形成了。

对于这种单表死锁，主要是由于索引资源的争夺造成的，所以只需要建立正确的索引便可避免死锁的发生。查询 B 中是因为 c3 字段不在索引中，所以需要请求对聚集索引的共享锁以查找 c3 值，所以具体的解决办法就是将 c3 字段包括在非聚集索引中便可，重新创建非聚集索引的脚本如代码 21.28 所示。

代码 21.28　重新创建非聚集索引

```
DROP INDEX t1.IX_t1_c2
GO
CREATE INDEX IX_t1_c2    --创建包含索引
ON t1(c2)
INCLUDE(c3)              --将 c3 字段包含在索引中
```

重新建立索引后接下来再运行查询 A 和查询 B，将不会再发生死锁了。

21.4.5　死锁监视与预防

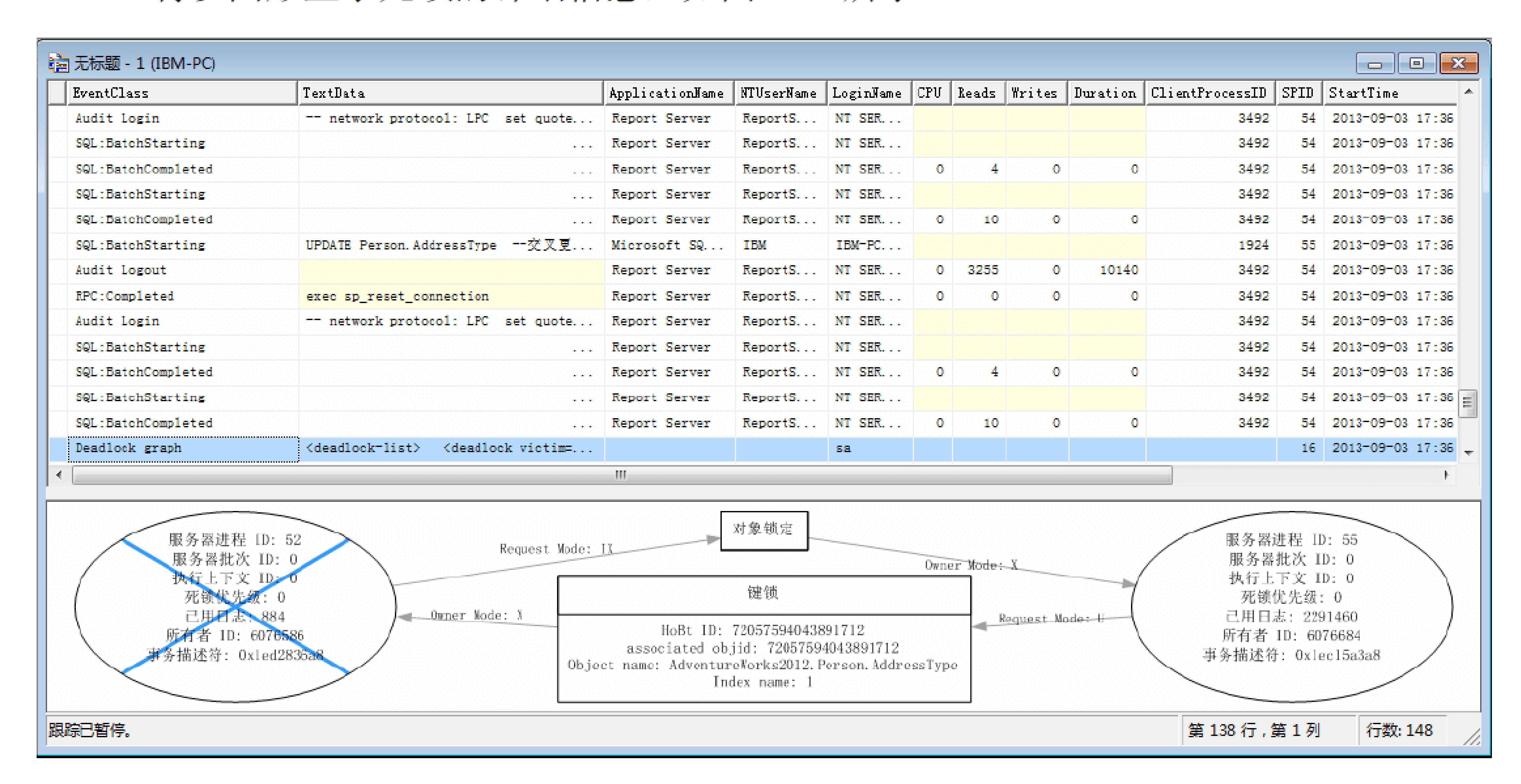
发生死锁的时候最直观的表现就是系统抛出 1205 异常，将某个事务作为牺牲品。但是单独从这个异常并不能判断和获得死锁中争夺的资源，请求的锁类型等重要的信息。使用 DBCC 命令或者是 SQL Server Profiler 可以获得死锁发生时的详细信息。

在 SQL Server 2012 中，使用以下 DBCC 命令可以让 SQL Server 将死锁信息记录到数据库的日志中。

```
dbcc traceon(1222,1204,3605,-1)
```

使用 DBCC 命令主要是用于死锁偶尔出现的系统，一旦死锁发生，系统就可以将详细死锁信息记录在 SQL Server 日志中，以便用户查看。

对于死锁频繁发生的系统，则使用 Profiler 进行跟踪更方便。Profiler 可以以图形的方式来显示死锁发生时的详细信息。在开启 Profiler 时，在 Locks 事件下选中 Deadlock graph 选项设置跟踪死锁并以图形显示。启用 Profiler 跟踪后，在 SSMS 中运行前面的死锁示例，Profiler 将以图形显示死锁的详细信息，如图 21.2 所示。



无标题 - 1 (IBM-PC)

EventClass	TextData	ApplicationName	NTUserName	LoginName	CPU	Reads	Writes	Duration	ClientProcessID	SPID	StartTime
Audit Login	-- network protocol: LPC set quote...	Report Server	ReportS...	NT SER...					3492	54	2013-09-03 17:36
SQL:BatchStarting	...	Report Server	ReportS...	NT SER...					3492	54	2013-09-03 17:36
SQL:BatchCompleted	...	Report Server	ReportS...	NT SER...	0	4	0	0	3492	54	2013-09-03 17:36
SQL:BatchStarting	...	Report Server	ReportS...	NT SER...					3492	54	2013-09-03 17:36
SQL:BatchCompleted	...	Report Server	ReportS...	NT SER...	0	10	0	0	3492	54	2013-09-03 17:36
SQL:BatchStarting	UPDATE Person.AddressType --交叉更...	Microsoft SQ...	IBM	IBM-PC...					1924	55	2013-09-03 17:36
Audit Logout		Report Server	ReportS...	NT SER...	0	3255	0	10140	3492	54	2013-09-03 17:36
RPC:Completed	exec sp_reset_connection	Report Server	ReportS...	NT SER...	0	0	0	0	3492	54	2013-09-03 17:36
Audit Login	-- network protocol: LPC set quote...	Report Server	ReportS...	NT SER...					3492	54	2013-09-03 17:36
SQL:BatchStarting	...	Report Server	ReportS...	NT SER...					3492	54	2013-09-03 17:36
SQL:BatchCompleted	...	Report Server	ReportS...	NT SER...	0	4	0	0	3492	54	2013-09-03 17:36
SQL:BatchStarting	...	Report Server	ReportS...	NT SER...					3492	54	2013-09-03 17:36
SQL:BatchCompleted	...	Report Server	ReportS...	NT SER...	0	10	0	0	3492	54	2013-09-03 17:36
Deadlock graph	<deadlock-list> <deadlock victim=...			sa						16	2013-09-03 17:36

图 21.2　Profiler 图形显示死锁

尽管死锁不能完全避免，但遵守特定的编码惯例可以将发生死锁的机率降至最低。死锁减少可以减小资源被锁定的时间，增加业务的吞吐量。同时还可以减小系统开销。下列方法有助于减少死锁发生的可能性。

- 按同一顺序访问对象。大部分死锁都是因为访问对象的顺序不一致造成的。如果所有并发事务按同一顺序访问对象，则发生死锁的可能性会降低。
- 避免事务中的用户交互。因为没有用户干预的批处理的运行速度远快于用户必须手动响应查询时的速度。如果开启事务后再与用户交互，而用户并没有提交事务，则事务将一直等待用户提交而一直占用其他资源。
- 事务尽量简短。事务运行的时间长，则占用资源的时间就长，其持有排他锁或更新锁的时间也就越长，从而会阻塞其他活动并可能导致死锁。
- 使用较低的隔离级别。在单表死锁小节中的死锁就发生在高隔离级别的事务中，而如果使用较低的事务隔离级别，如未提交读或已提交读，则不会造成死锁。
- 建立正确的索引。索引建立的不正确也会造成死锁的发生。另外正确的索引还可以提高事务访问数据的速度，减少事务占用资源的时间。
- 让不同的连接使用相同的锁定。使用绑定连接，同一应用程序打开的两个或多个连接可以相互合作。可以像主连接获取的锁那样持有次级连接获取的任何锁，反之亦然。这样它们就不会互相阻塞。

21.5　小　　结

本章主要讲解了 SQL Server 中事务处理的相关知识。事务处理中最重要的两个概念就是锁和隔离级别。

锁定是 SQL Server 数据库引擎用来同步多个用户同时对同一个数据块访问的一种机制。锁分为多个模式来锁定资源，不同的模式之间存在不同的兼容性。对资源的锁定具有不同的粒度，小到锁定一个行，大到锁定整个数据库。

在并发操作的情况下，可能造成脏读、不可重复读、幻读等负面影响。为了避免这些负面影响，SQL Server 采用隔离级别对资源访问和数据更改进行隔离。同时并发操作还有可能造成两个或多个事务之间相互锁定资源的情况形成死锁。可以通过多种方法来监视和预防死锁。

第 22 章　数据库系统调优工具

在数据库性能调优中首先要发现性能瓶颈，然后才能针对具体的瓶颈进行优化，本章将主要介绍如何使用 SQL Server 和 Windows 提供的工具和方法，进行性能调优。性能是决定数据库读取速度的关键，在进行应用程序开发和数据库设计时，读取速度是设计人员首先要考虑的问题。

22.1　数据库报表

SQL Server 中的报表功能不仅可以用于在报表服务中展示业务数据，在 SSMS 中也可以使用报表来展示数据库相关的数据。

22.1.1　查看数据库实例报表

SQL Server 2012 中直接为每个数据库实例提供了多个标准报表用于显示该数据库实例中的配置更改历史记录、内存占用、活动情况、性能情况等。SQL Server 2012 中默认提供的标准报表如图 22.1 所示。

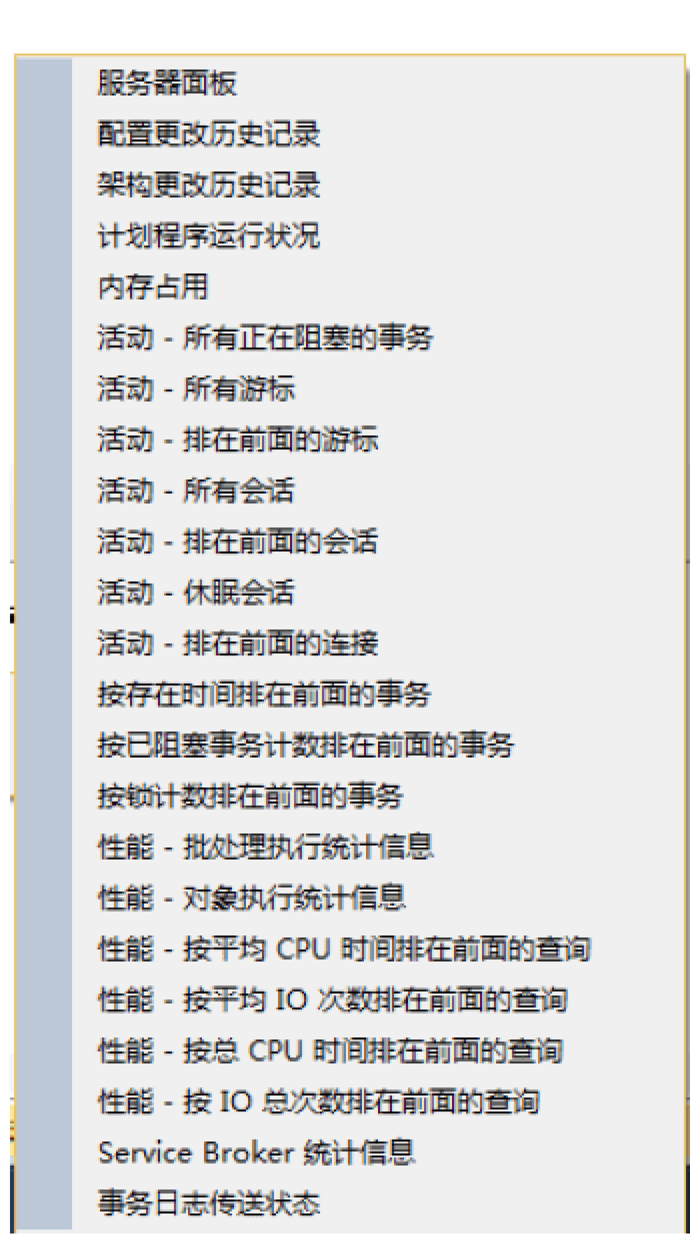


图 22.1　数据库实例标准报表

例如要查看当前数据库实例中内存的分配占用情况，则需要在对象资源管理器中右击“数据库实例”图标，在弹出的快捷菜单中选择“报表”|“标准报表”|“内存占用”命令，系统将列出当前数据库实例的内存占用报表，如图 22.2 所示。

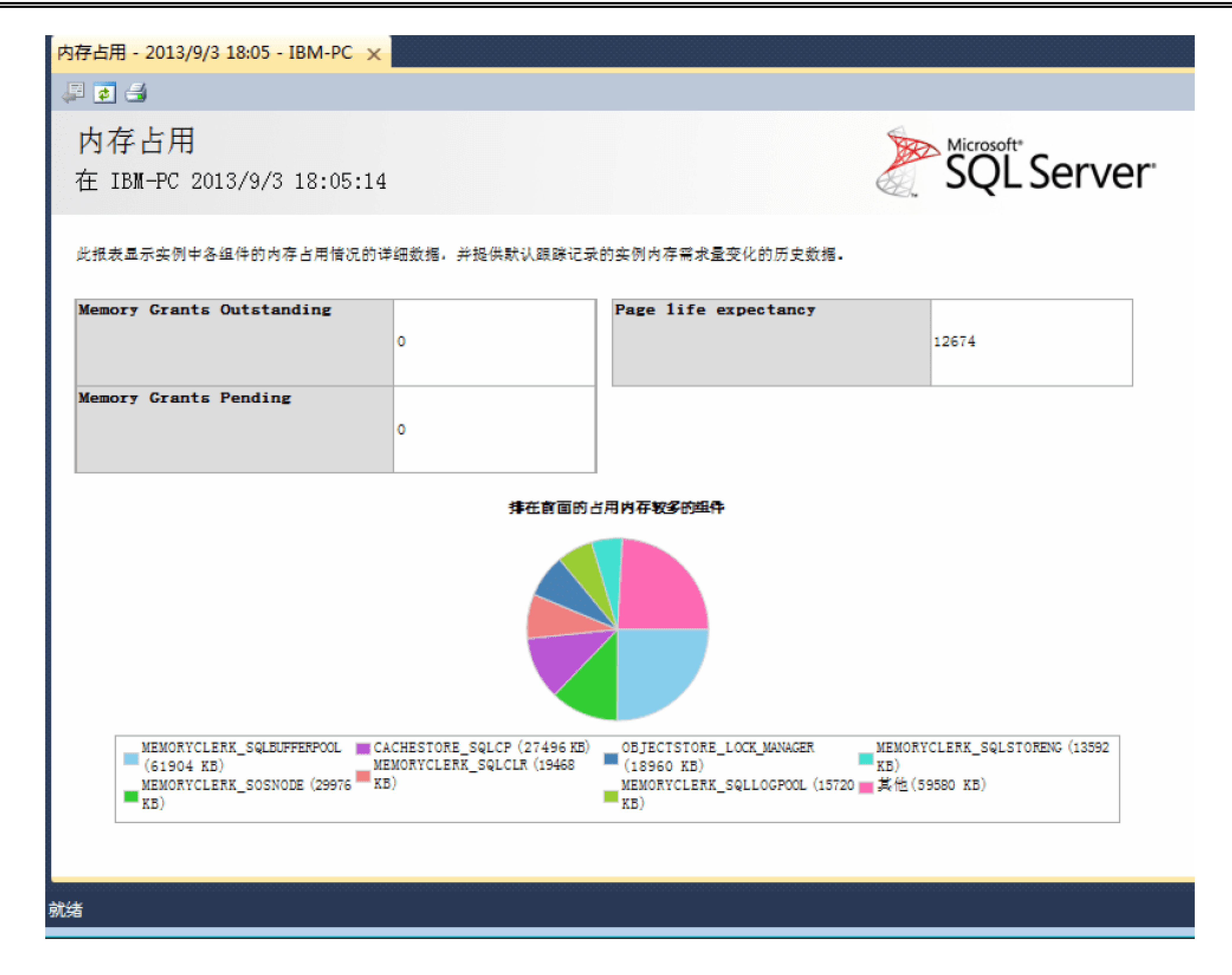


图 22.2　数据库实例内存占用报表

这些报表是使用 SQL Server 的报表设计器做出来的，但是并不需要报表服务的支持，也不需要 IIS 的支持，而是直接集成到 SSMS 中的。

这些报表中的数据都来自 SQL Server 数据库，也就是说，使用对应的 T-SQL 查询也可以得出相同的结果。只不过数据库报表相对直观，图文并茂，而且不需要编写任何脚本，只需要鼠标单击即可。

22.1.2　查看单个数据库报表

除了数据库实例的报表外，SQL Server 2012 默认还提供了多个标准报表用于展示数据库的磁盘、索引、事务等相关信息。SQL Server 2012 默认提供的数据库标准报表如图 22.3 所示。

若需要查看某个数据库的每个表中的数据和占用的磁盘空间，可以在 SSMS 中右击该数据库，然后在弹出的快捷菜单中选择“报表”|“标准报表”|“按表的磁盘使用情况”命令。系统将在 SSMS 中列出该数据库所有用户表的记录数、保留大小、数据大小、索引大小和未使用空间大小，如图 22.4 所示。

使用数据库报表可以快速列出常用的一些性能信息，帮助用户查看与比较各数据库对象的属性。

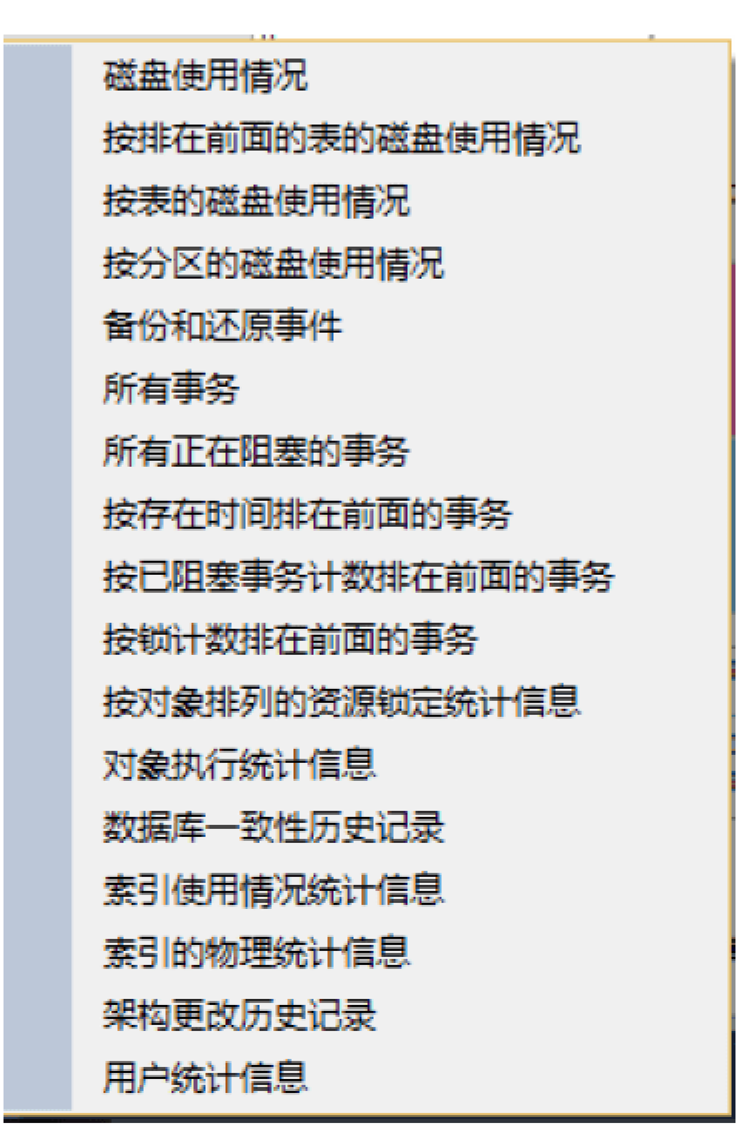
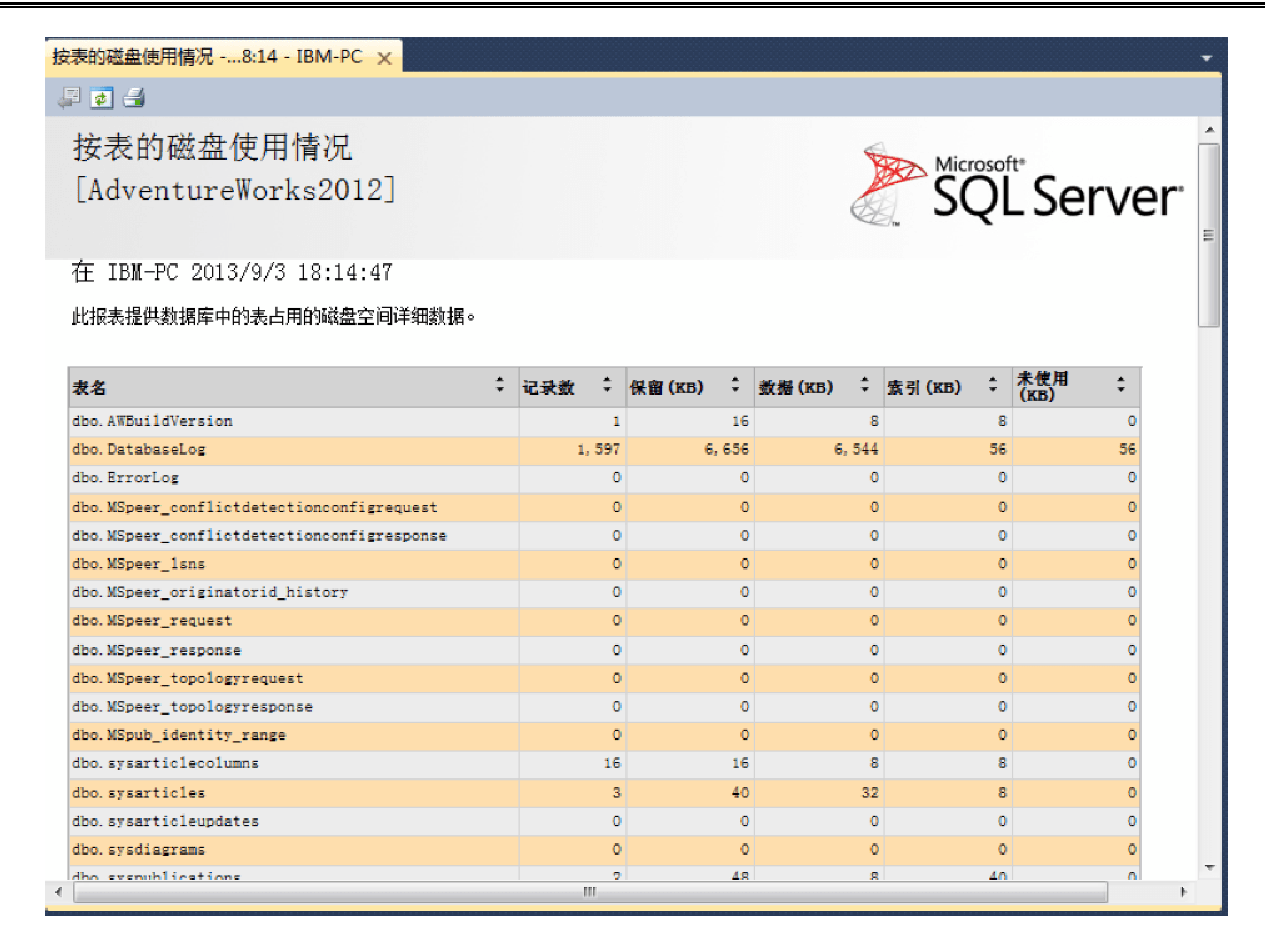


图 22.3　数据库标准报表



表名	记录数	保留 (KB)	数据 (KB)	索引 (KB)	未使用 (KB)
dbo.AWBuildVersion	1	16	8	8	0
dbo.DatabaseLog	1,597	6,656	6,544	56	56
dbo.ErrorLog	0	0	0	0	0
dbo.MSpeer_conflictdetectionconfigrequest	0	0	0	0	0
dbo.MSpeer_conflictdetectionconfigresponse	0	0	0	0	0
dbo.MSpeer_lsns	0	0	0	0	0
dbo.MSpeer_originatorid_history	0	0	0	0	0
dbo.MSpeer_request	0	0	0	0	0
dbo.MSpeer_response	0	0	0	0	0
dbo.MSpeer_topologyrequest	0	0	0	0	0
dbo.MSpeer_topologyresponse	0	0	0	0	0
dbo.MSpub_identity_range	0	0	0	0	0
dbo.sysarticlecolumns	16	16	8	8	0
dbo.sysarticles	3	40	32	8	0
dbo.sysarticleupdates	0	0	0	0	0
dbo.sysdiagrams	0	0	0	0	0

图 22.4　数据库报表

说明： 在 SQL Server 2005 中还专门提供了用于性能分析的一套报表 SQL Server 2005 Performance Dashboard，需要单独从官方网站下载安装。在 SQL Server 2008 以后的版本中则集成了“性能和监视器”功能实现了类似的报表，所以就不再提供 SQL Server 2008 Performance Dashboard 了。

22.2　使用 SQL Server Profiler 跟踪数据库

SQL Server Profiler（以下就简称 Profiler）是跟踪数据库执行的工具。通过该工具可以跟踪到每时每刻数据库引擎执行操作产生的异常等信息，是性能调优中必不可少的一个工具。

22.2.1　创建 SQL Server Profiler

通过选择开始菜单在 Microsoft SQL Server 2012 性能工具目录下的 SQL Server Profiler 命令可以打开 Profiler。如果当前已经打开了 SSMS，也可以通过选择 SSMS 的“工具”菜单下的 SQL Server Profiler 选项来打开。

在运行 Profiler 之后，与 SSMS 一样，系统要求输入要登录的服务器和验证信息。在输入用户名密码并验证通过后，将弹出“跟踪属性”对话框，如图 22.5 所示。

其中“跟踪名称”文本框中输入的是本次跟踪的名字，用户可以修改，也可以使用默认值。Profiler 提供了模板功能，不同的模板对应跟踪不同的对象和事件，如果需要自定义

跟踪的对象和属性，则选择使用默认值，在“事件选择”选项卡中再修改具体要跟踪的事件和列。

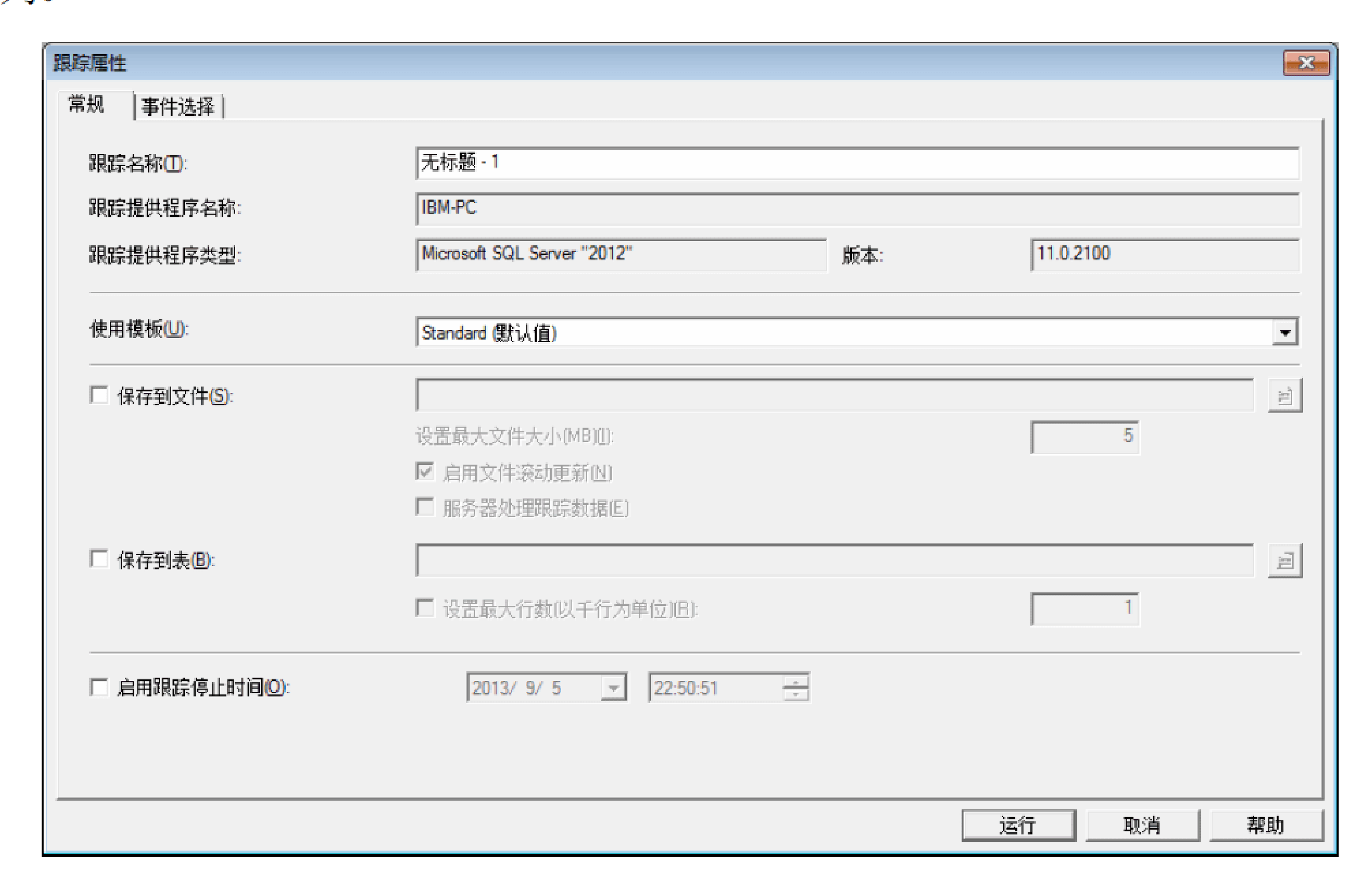


图 22.5　Profiler 的跟踪属性对话框

Profiler 允许将跟踪的结果保存为文件，也可以直接将跟踪结果保存为数据库中的表。保存为文件的话由于是直接进行磁盘读写，所以效率较高，对跟踪的数据库影响较小。保存到表的话则最好不要将跟踪保存到被跟踪的数据库实例中，否则将增加被跟踪数据库的负担。将跟踪保存到表的优点就是可以通过编写 T-SQL 语句来查询跟踪记录。

保存到文件中的跟踪记录也可以再导入到数据库表中，所以一般在跟踪时使用“保存到文件”复选框，跟踪完成后再将跟踪的记录导入到数据库表中，使用 T-SQL 语句来分析跟踪数据。如果既不保存到文件也不保存到表，则 Profiler 会使用一个临时文件来保存跟踪记录，在 Profiler 关闭时也清除掉该临时文件。这里选择将跟踪保存到 C 盘。

在 Profiler 的跟踪属性中还可以配置跟踪停止的时间，到了该时间后跟踪将自动停止。选择“事件选择”选项卡，切换到事件选择界面，如图 22.6 所示。

选中“显示所有事件”复选框，将在事件列表中列出所有可以被跟踪的事件。在性能调优过程中则需要根据实际情况来选择事件，对于常见情况，一般只跟踪存储过程和 T-SQL 语句的执行情况即可，即 RPC:Completed 事件和 SQL:BatchCompleted 事件。将这两个事件前面的复选框选中，取消选中其他事件前的复选框。

选中“显示所有列”复选框，将列出选中事件所有可被跟踪的列，例如 Database Name 列、Error 列等。如果被跟踪的数据库实例中具有多个数据库，一般需要将 DatabaseName 列选中，以区分不同的数据库执行的存储过程和脚本。

在数据库性能调优时，Profiler 中重点关注的列如下。

- TextData：被跟踪的脚本内容。
- CPU：事件使用的 CPU 时间，单位是毫秒。
- Reads：事件读取磁盘的次数，该值在性能调优中相当重要。
- Writes：事件写入磁盘的次数。

- Duration：执行事件时花费的时间，单位是毫秒。

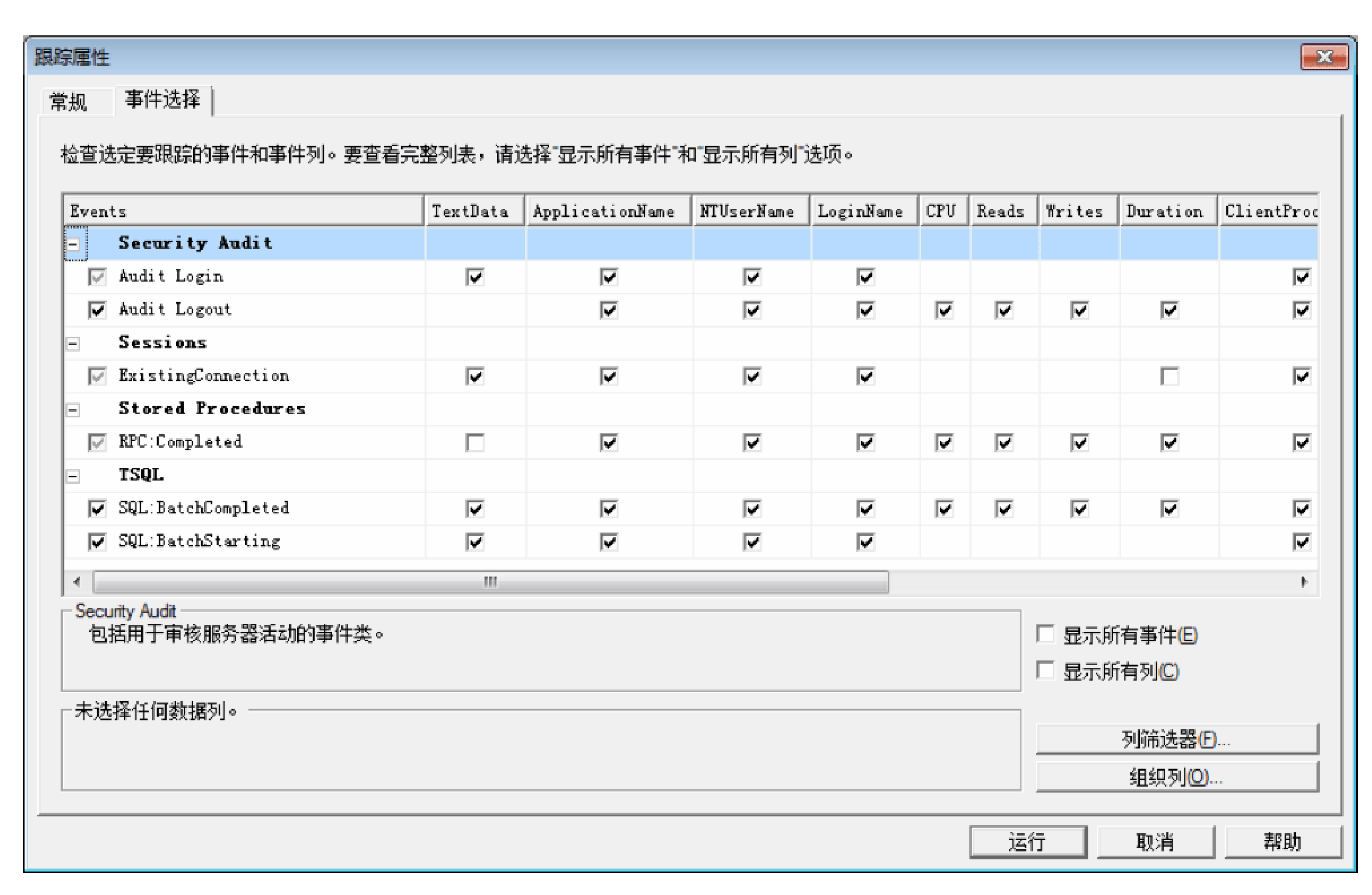


图 22.6　Profiler 的事件选择

Profiler 默认情况下是跟踪了整个数据库实例的事件，如果多个数据库在被使用，而我们只需要跟踪其中的一个数据库，这时可以使用列筛选器。单击“列筛选器”按钮，弹出筛选器的编辑对话框，如图 22.7 所示。

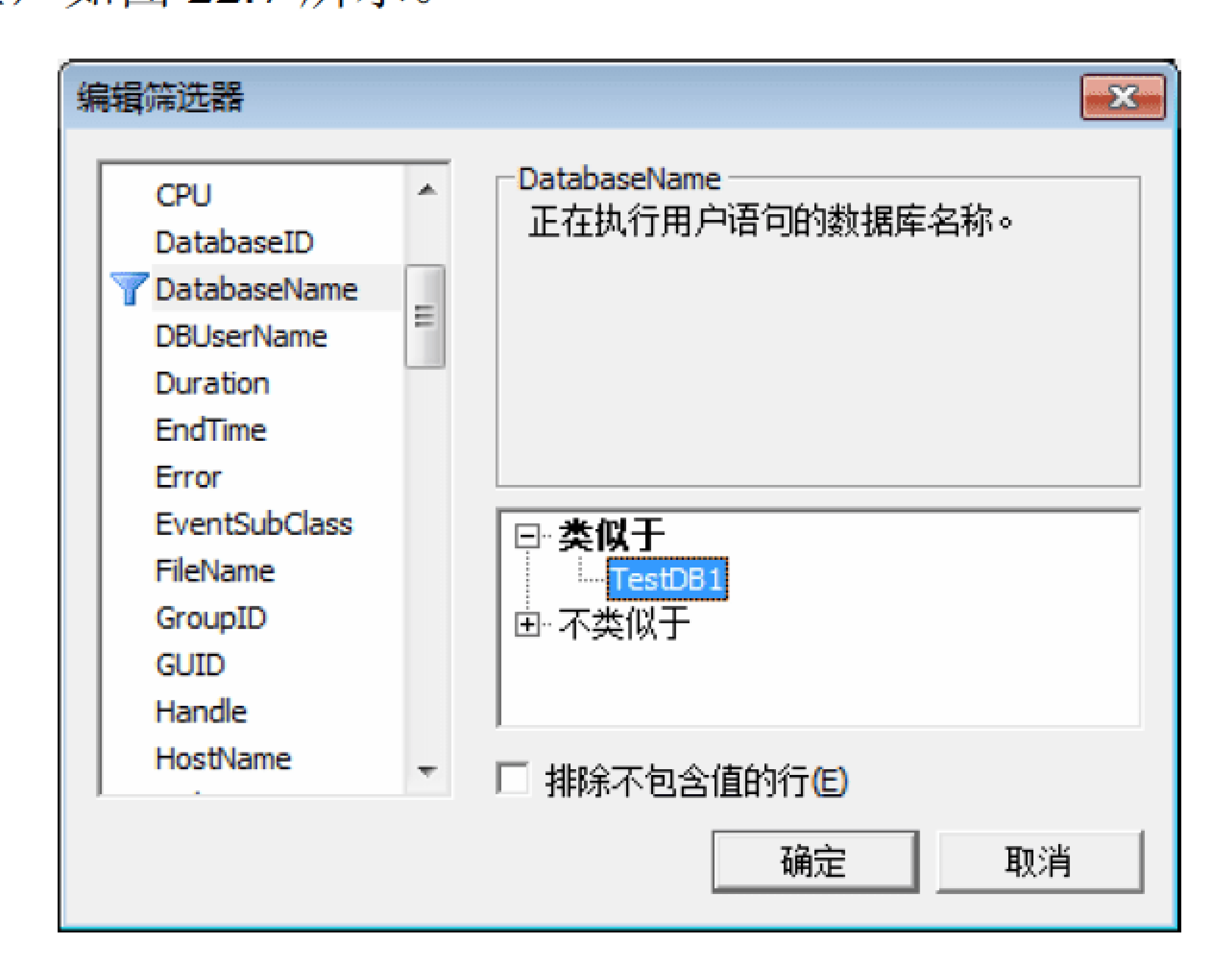


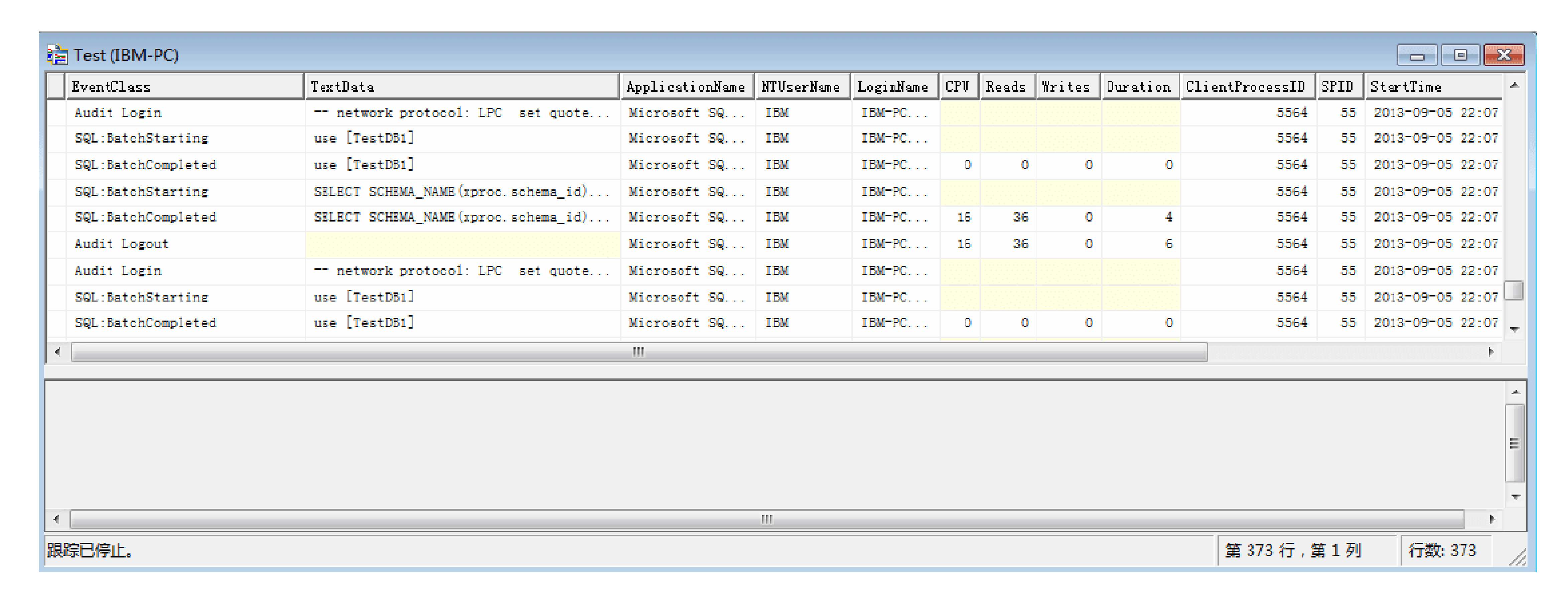
图 22.7　“编辑筛选器”对话框

例如现在只希望跟踪 TestDB1 中的存储过程和脚本的执行情况，其他数据库不跟踪，则选择 DatabaseName 列，然后编辑“类似于”节点的值为 TestDB1。“排除不包含值的行”复选框是指如果跟踪的数据库中没有 DatabaseName 列的值则排除。单击“确定”按钮即可完成列筛选器的设置，回到 Profiler 跟踪属性窗口。“组织列”按钮可以设置被跟踪事件列的顺序。

一切配置完成，单击“运行”按钮，Profiler 将启动对数据库实例的事件跟踪。

22.2.2　查询 SQL Server Profiler

在启动跟踪后，Profiler 将实时跟踪并显示跟踪到的事件。在 SSMS 中运行几个简单的查询，然后再观察 Profiler，将可以看到运行的查询已经被 Profiler 记录下来，如图 22.8 所示。



EventClass	TextData	ApplicationName	NTUserName	LoginName	CPU	Reads	Writes	Duration	ClientProcessID	SPID	StartTime
Audit Login	-- network protocol: LPC set quote...	Microsoft SQ...	IBM	IBM-PC...					5564	55	2013-09-05 22:07
SQL:BatchStarting	use [TestDB1]	Microsoft SQ...	IBM	IBM-PC...					5564	55	2013-09-05 22:07
SQL:BatchCompleted	use [TestDB1]	Microsoft SQ...	IBM	IBM-PC...	0	0	0	0	5564	55	2013-09-05 22:07
SQL:BatchStarting	SELECT SCHEMA_NAME(xproc.schema_id)...	Microsoft SQ...	IBM	IBM-PC...					5564	55	2013-09-05 22:07
SQL:BatchCompleted	SELECT SCHEMA_NAME(xproc.schema_id)...	Microsoft SQ...	IBM	IBM-PC...	16	36	0	4	5564	55	2013-09-05 22:07
Audit Logout		Microsoft SQ...	IBM	IBM-PC...	16	36	0	6	5564	55	2013-09-05 22:07
Audit Login	-- network protocol: LPC set quote...	Microsoft SQ...	IBM	IBM-PC...					5564	55	2013-09-05 22:07
SQL:BatchStarting	use [TestDB1]	Microsoft SQ...	IBM	IBM-PC...					5564	55	2013-09-05 22:07
SQL:BatchCompleted	use [TestDB1]	Microsoft SQ...	IBM	IBM-PC...	0	0	0	0	5564	55	2013-09-05 22:07

图 22.8　Profiler 跟踪结果

注意： 在 SSMS 中执行操作时很容易产生各种操作脚本形成干扰，但是在跟踪实际项目数据库时，由于是利用 ADO.NET 来连接数据库，所以并不会产生大量的干扰跟踪脚本。

在 Profiler 的工具栏中提供了开始、暂停和停止跟踪的按钮，用户可以随时暂停、停止跟踪或重新启动跟踪。用户也可以随时单击“清除”按钮清除已有的跟踪记录。在跟踪子窗口中通过上下两栏来显示跟踪结果，上栏为跟踪记录列表，下栏则是对应选中的跟踪记录的脚本。

在执行了一系列不同的 SQL 查询后，Profiler 中也记录了大量的跟踪记录。单击“停止”按钮即可停止当前的跟踪。选择“文件”|“另存为”|“跟踪表”命令，弹出数据库登录对话框，在连接到要保存跟踪表的数据库后，弹出目标表设置对话框，如图 22.9 所示。

目标表
选择用于跟踪的目标表。
SQL Server: IBM-PC
数据库(D): TestDB1
架构(S): dbo
表(T): Student
确定　取消　帮助

图 22.9　跟踪数据保存的表

选择要保存跟踪数据的数据库、架构和表，然后单击“确定”按钮，Profiler 将会把所有的跟踪数据导入到该表中，接下来即可通过 T-SQL 来查询分析跟踪数据。

例如要查询读取页数最多的前 10 条跟踪记录，则对应的查询脚本如代码 22.1 所示。

代码 22.1　查询读页数最多的跟踪记录

```
SELECT TOP 10 TextData,Reads,Writes,Duration
FROM Track1
ORDER BY Reads DESC --根据读页数的多少从大到小排序
```

如果要查询根据执行时间和写页数排序等跟踪记录同样只需要编写相应的 T-SQL 语句即可。

22.3　性能监视器

Windows 中提供了性能监视器（Performance Monitor），用于查看和跟踪系统资源及对象属性的使用和变化情况。性能监视器可以协助判断系统性能的瓶颈，为数据库的性能优化找准方向。

22.3.1　性能监视器简介

性能监视器是 Windows 操作系统自身的监视工具，可以实时跟踪当前系统的各个性能指标或查看历史数据。可以通过拖放或创建自定义数据收集器集将性能计数器添加到性能监视器中。使用性能监视器可以直观地查看性能日志数据的多个图表视图。可以在性能监视器中创建自定义视图，该视图可以导出为数据收集器集，以便与性能和日志记录功能一起使用。

在“开始”菜单下的“运行命令”文本框中输入 perfmon 或者在“管理工具”菜单中选择“性能”选项，将打开“性能监视器”对话框，如图 22.10 所示。

“性能监视器”对话框左侧为控制台根节点，相当于 SSMS 中的对象资源管理器，在性能监视器中提供了系统监视器、计数器日志、跟踪日志和警报。右侧则是性能计数器的展示页面，该界面分为上下栏，上栏通过坐标曲线展示了计数器中的跟踪数据，下栏则列出了当前启用的计数器及其相关属性。

在 Windows 7 中，默认情况下，系统在性能监视器中只有 1 个计数器% Processor Time。它是所有进程线程使用处理器执行指令所花的时间百分比。指令是计算机执行的基础单位。线程是执行指令的对象，进程是程序运行时创建的对象。此计数包括处理某些硬件间隔和陷阱条件所执行的代码。

通过单击计数器上方工具栏中的“添加”按钮可以为性能监视器添加更多的计数器。单击“添加”按钮后弹出“添加计数器”对话框，如图 22.11 所示。

性能监视器不仅可以监视本机性能，也可以监视网络上其他计数器的性能，选中“从计算机选择计数器”单选按钮，可以在下拉列表中选择要监视的计算机。

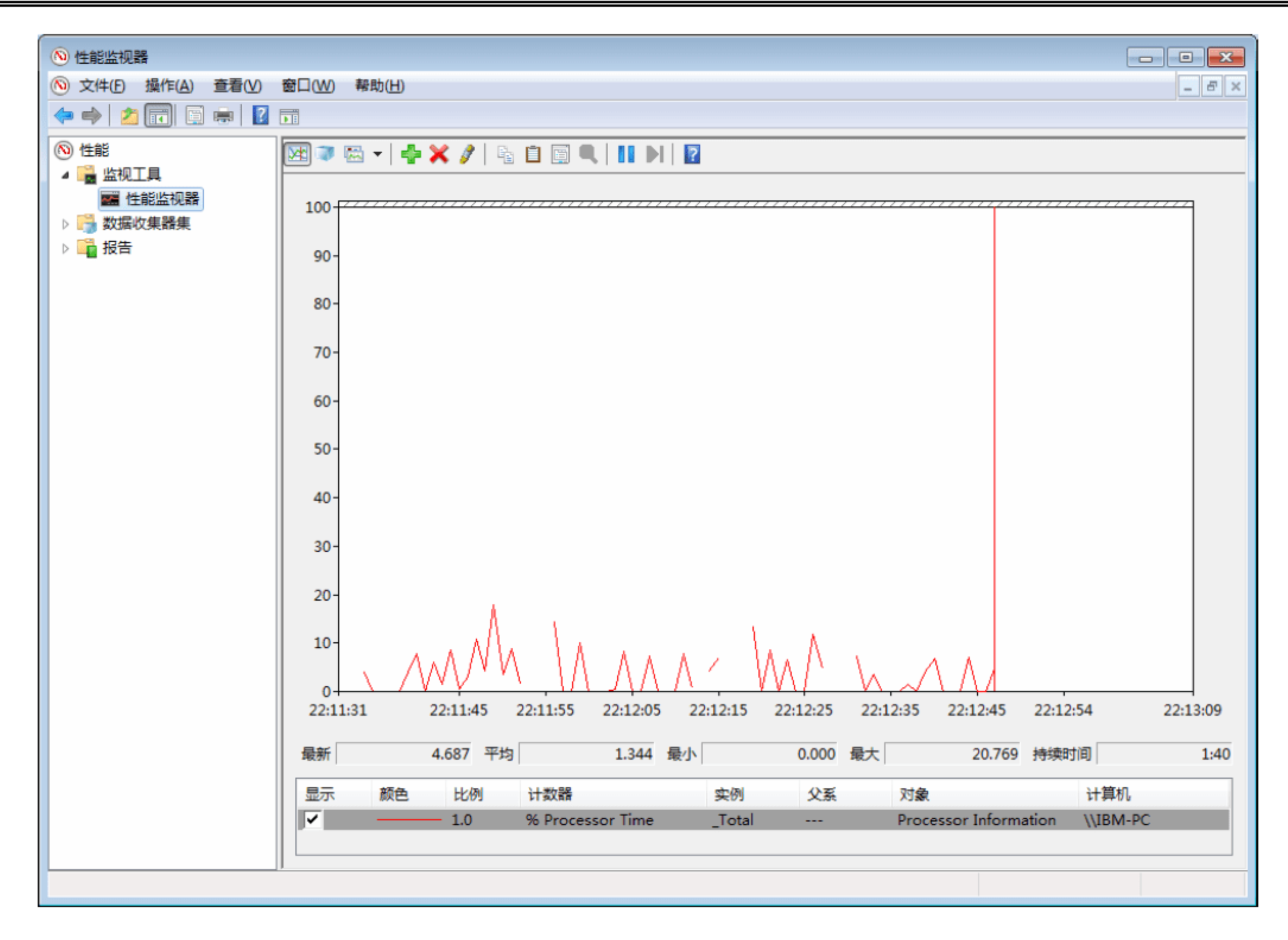


图 22.10 “性能监视器”对话框

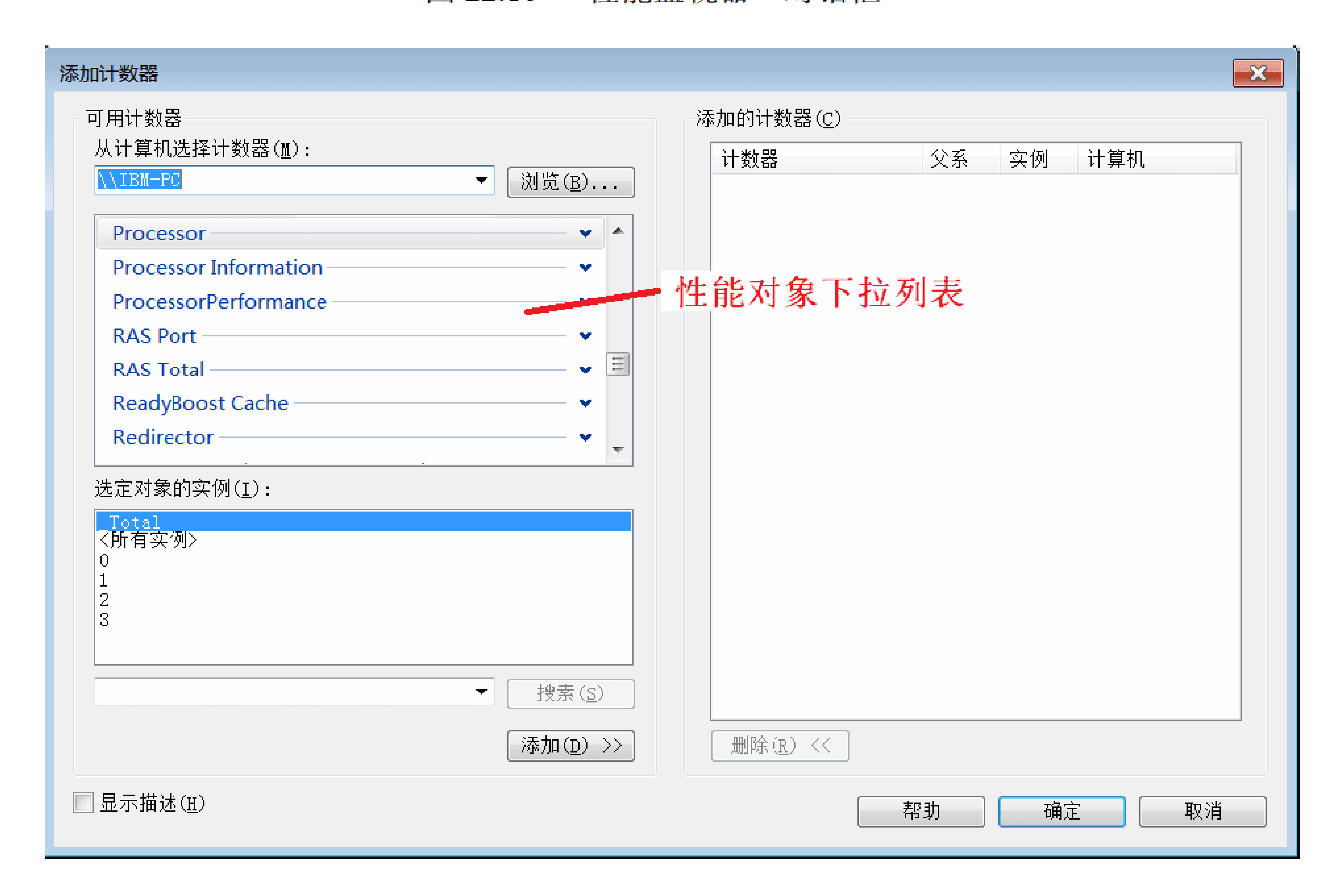


图 22.11 “添加计数器”对话框

“性能对象”下拉列表中列出了选中的计算机上所有的性能对象，除了操作系统自带的性能对象外，如果安装了.NET 则还会有对应.NET 的性能对象，安装了 SQL Server 将可以看到对应的 SQL Server 性能对象。其他的软件也可能添加自己的性能对象。

选择好性能对象后，下方将列出该性能对象下的所有计数器，可以选择“所有计数器”

单选按钮，将该性能对象下的所有计数器添加到监视器中。也可以选择具体的计数器，然后在右侧选择对应的实例，再单击“添加”按钮即可将选中的计数器添加到监视器中。

22.3.2　常用的计数器

系统默认的计数器并不足以帮助用户判断性能瓶颈，用户需要在监视器中添加更多的性能计数器。就 SQL Server 的特性来说，按照重要程度，观察硬件资源不足的顺序依次是：内存、硬盘、CPU 和网络。SQL Server 是一个非常耗内存的程序，它将尽量占用可用的内存以缓存更多的数据，从而提高数据处理的效率。

🔔注意：默认情况下，32 位的环境中 SQL Server 最多只能占用 2G 的内存，如果服务器内存充足则有必要开启 3GB 开关或 AWE，以使 SQL Server 能够使用更多的内存作为缓存，从而提高性能。

当内存不足的时候，将会影响到硬盘和 CPU。所以在性能监视器中看到 CPU 或硬盘的计数器超标，不一定性能瓶颈就在 CPU 或硬盘上，有可能源头是内存不足造成的。

硬盘是存取数据库数据和日志的地方，另外也是虚拟内存所在地。服务器中最好将虚拟内存、数据文件和日志文件分开放在不同的硬盘上，因为这 3 种文件的设计目的不同，访问频率也不同，如果都放在一个硬盘上，大量的数据读写将会导致系统同时对这 3 种文件进行大量的随机读取，而不是顺序读取，使得硬盘的性能下降。

CPU 作为计算机的大脑，其重要性不言而喻，它的快慢直接关系到整个系统数据处理的快慢。在多数服务器中，CPU 的处理速度极快，所以一般不容易在 CPU 上产生瓶颈，在监视到 CPU 占用率过高时，多数情况都是由于 I/O 数据量太大造成的。如表 22.1 列出了在处理器、内存、磁盘和网络这 4 个方面常用的一些计数器。

表 22.1　常用的计数器

系统资源	监视的目的	性能项目/计数器	说　明
处理器	使用信息	Processor\%Processor Time	CPU的利用率
	瓶颈	System\Processor Queue Length	处理器拥有的线程数
		Processor\%Interrupts/sec	每秒处理器处理的硬件超出的平均值
		System\Context switches/sec	包含了从任意线程向其他线程转换的全部处理器的比率
内存	使用状况	Memory\Available Bytes	运行中的程序利用的物理内存的大小
		Memory\Cache Bytes	文件系统缓冲正在使用的比特数
	瓶颈	Memory\Page/sec	为了解决硬盘页错误从磁盘读取，或写入磁盘的速度
		Memory\Page Faults/sec	每秒钟出错页面的平均数量
	内存泄漏	Memory\Page Input/sec	从磁盘读取页面以解析硬盘页面错误的速度
		Memory\Page Reads/sec	为了解决硬盘页失误磁盘被读的页数
		Memory\Transition Faults/sec	是恢复页面解析页面错误的速度
		Memory\Pool Paged Bytes	指在分页池中的字节数
		Memory\Pool Nonpaged Bytes	指在非分页池中的字节数

续表

系统资源	监视的目的	性能项目/计数器	说　明
磁盘	使用状况	LogicalDisk\% Free Space(*1)	空间（%）
		LogicalDisk\% Disk Time(*1)	访问时间（%）
		PhysicalDisk\% Read/sec 1	在1秒内的读入动作回数
		PhysicalDisk\% Writes/sec 1	在1秒内的写入动作回数
	瓶颈	LogicalDisk\%Avg Disk Queue Length(*1)	指磁盘逻辑读取和逻辑写入请求队列的平均数
		PhysicalDisk\%Avg Disk Queue Length	指磁盘物理读取和物理写入请求队列的平均数
网络	使用状况	Network Segment\%Net Utilization(*2)	网络分割的利用率
	容许量	Network Interface\Bytes Total/sec	在1秒内NIC上被接收收信的比特数
		Network Interface\Packets/sec	在1秒内NIC上被接收送信的比特数
		Server\Bytes Total/sec	在1秒内服务器在网络间接收送信的比特数

22.3.3　计数器日志

计数器与 Profiler 一样可以将监视日志中的数据保存为文件，也可以保存到数据库中用于分析。计数器日志还可以设置启动和停止的时间，另外还可以设置在停止以后执行的命令。为了演示如何添加计数器日志，这里在 Windows Server 2003 环境下的性能监视器界面完成。

在性能监视器中添加计数器日志的操作如下。

（1）选择左侧的“计数器日志”节点，在弹出式菜单中选择“新建日志设置”选项，弹出对话框，要求输入新日志的名称。

（2）这里建一个测试日志，命名为 Test1，单击“确定”按钮弹出日志的设置对话框，如图 22.12 所示。

（3）单击“添加计数器”按钮，将需要记录的计数器添加到其中，数据采样间隔中可以设置每隔多长时间采集一次数据。这个采样间隔需要根据实际的情况而定，如果要记录的时间很长，则可以加大间隔。

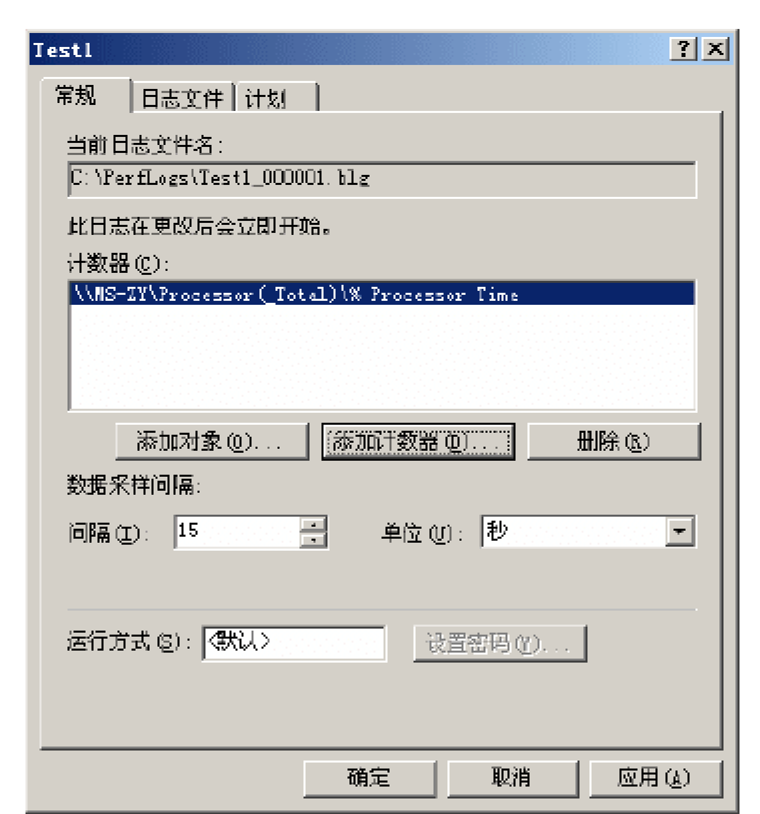
（4）选择“日志文件”选项卡，切换到日志文件类型和名称的设置界面，如图 22.13 所示。日志文件可以保存为二进制文件，文本文件也可以保存到 SQL Server 数据库中。为了便于查看，可以保存为文本文件，单击“配置”按钮可以配置文件的位置和大小限制。

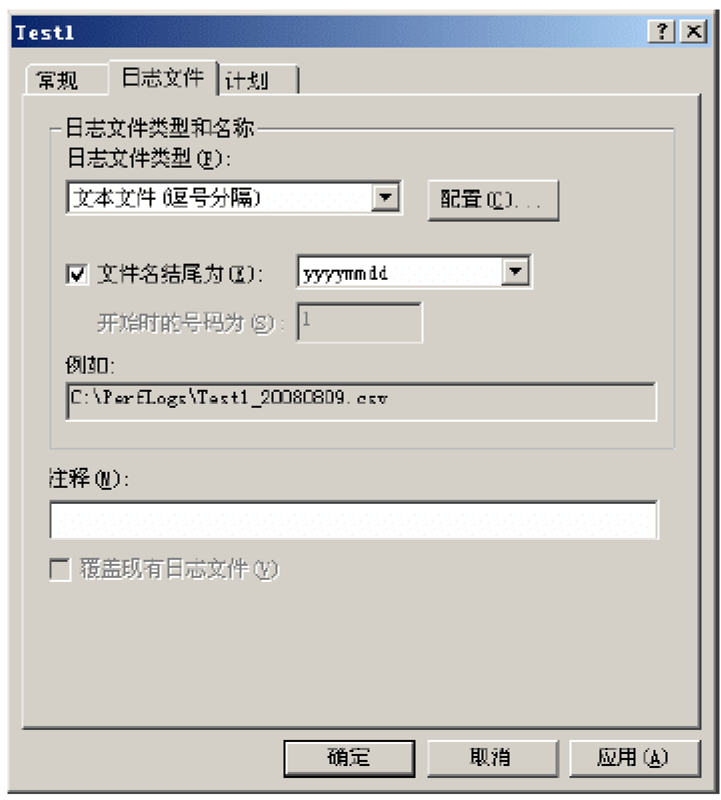
（5）“文件名结尾为”下拉列表框用于设置日志文件的结尾类型，而下方则给出了类型的示例。日志文件不一定只记录在一个文件中，可以通过结尾判断日志文件的顺序。这里选择的是使用年月日的方式结尾。

（6）选择“计划”选项卡，切换到日志的启动和停止时间的设置界面，如图 22.14 所示。

“启动日志”选项区域中可以设置日志开始记录的时间，在“停止日志”选项区域中

则可以设置停止日志记录的时间和停止后运行的命令。





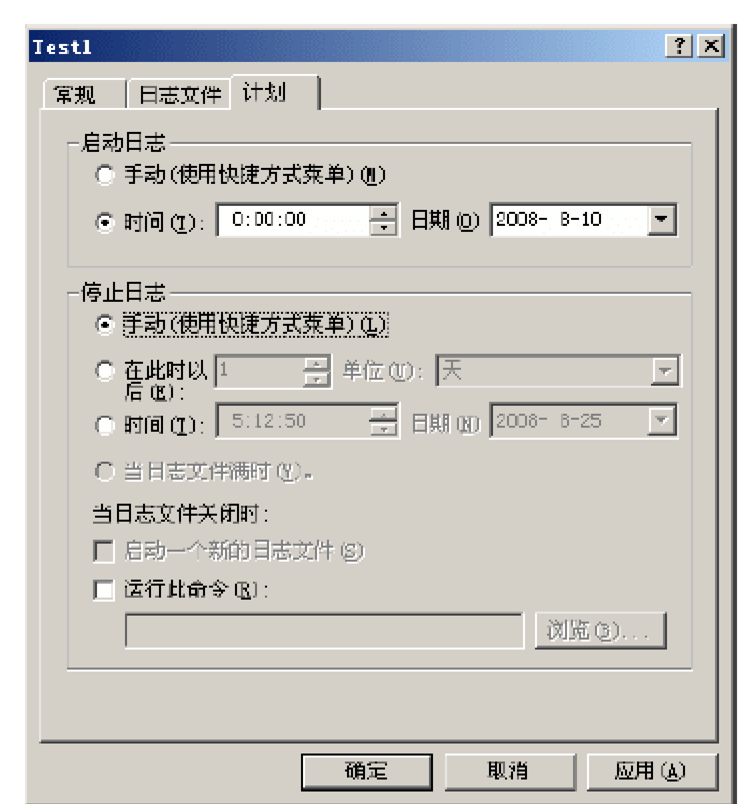


图 22.12　计数器日志设置　　图 22.13　日志文件类型和名称设置　　图 22.14　日志计划

（7）例如要记录某天晚上 12 点后系统的性能，则设置启动时间为晚上 0 点，停止日志可以设置为手动，也就是说日志启动后一直记录，直到人为停止为止。

（8）单击“确定”按钮即可完成计数器日志的创建。

记录的计数器日志可以使用性能监视器打开，如果是文本文件格式也可以直接使用文本编辑器打开。另外也可以将计数器日志文件中的数据导入到 SQL Server 中，使用 T-SQL 进行分析。

22.4　使用优化顾问优化 SQL 语句

在通过 Profiler 的跟踪，性能计数器的分析之后，可以找到最消耗资源的 SQL 查询，通常是由于使用不合理的索引造成了查询缓慢，通过使用数据库引擎优化顾问，可以帮助分析如何正确建立索引，从而提高查询的效率。

22.4.1　优化顾问简介

数据库引擎优化顾问（以下简称优化顾问）是 SQL Server 中自带的工具，用于分析在一个或多个数据库中运行的工作负荷的性能效果。工作负荷是对要优化的数据库执行的一组 T-SQL 语句。

在优化顾问分析了数据库的工作负荷效果后，会提供在 SQL Server 数据库中添加、删除或修改物理设计结构的建议。这些物理性能结构包括聚集索引、非聚集索引、索引视图和分区。如果用户选择实现这些建议，数据库引擎优化顾问使查询处理器能够用最短的时间执行工作负荷任务，提高查询的性能。

使用数据库引擎优化顾问优化数据库并不需要数据库结构、工作负荷或 SQL Server 内部工作方面的专业知识。

数据库引擎优化顾问提供了两种界面：

- 独立图形用户界面，它是一种用于优化数据库、查看优化建议和报告的工具。
- 命令行实用工具程序 dta.exe，用于实现数据库引擎优化顾问在软件程序和脚本方面的功能。

这里主要讲解图形用户界面的优化顾问的使用。借助数据库引擎优化顾问，用户不必精通数据库结构或深谙 Microsoft SQL Server，即可选择和创建索引、索引视图和分区的最佳集合。

在优化数据库时，优化顾问将使用跟踪文件、跟踪表或 T-SQL 脚本作为工作负荷输入。可以在 SSMS 中使用查询编辑器创建 T-SQL 脚本工作负荷，也可以通过使用 Profiler 中的优化模板来创建跟踪文件和跟踪表工作负荷。

使用数据库引擎优化顾问进行数据库优化，通过分析工作负荷提供了下列功能：

- 推荐数据库的最佳索引组合。
- 推荐对齐分区或非对齐分区。
- 推荐数据库的索引视图。
- 分析在应用建议的更改后将产生多大的性能提升。
- 推荐为执行一个小型的问题查询集而对数据库进行优化的方法。
- 允许通过指定磁盘空间约束等高级选项对推荐进行自定义。
- 提供对所给工作负荷的建议执行效果的汇总报告。
- 以假定配置的形式提供可能的设计结构方案，供数据库引擎优化顾问进行评估，从中推荐备选方案。

22.4.2　使用优化顾问优化 SQL 语句

以 AdventureWorks2012 数据库为例，代码 22.2 为在该数据库中的一个查询。

代码 22.2　读页数多的一个查询

```
USE AdventureWorks2012;
GO
SELECT SalesOrderID,SalesOrderDetailID,CarrierTrackingNumber,rowguid
FROM Sales.SalesOrderDetail
WHERE UnitPrice=33.7745
```

下面就使用优化顾问来优化该查询，具体操作如下。

（1）将查询 SQL 语句复制到 SSMS 中，执行 SET STATISTICS IO ON 语句打开 I/O 统计，运行该查询可以看到执行了 1241 次逻辑读取，这是优化前读取的页数。

（2）选择“查询”菜单中的“数据库引擎优化顾问中的分析查询”选项，打开优化顾问对话框，如图 22.15 所示。

（3）由于是在 SSMS 中启动的优化顾问，所以系统已经将查询优化所对应的数据库添加到优化对象中，通过数据库列表框可以选择查询对应的数据库。会话名称则使用的是当前用户名和时间，一般不需要修改。

（4）选择“优化选项”选项卡，切换到优化选项的设置界面，如图 22.16 所示。

（5）在优化选项中可以设置本次优化的语句使用的物理设计结构、分区策略及优化保

留的物理设计结构。

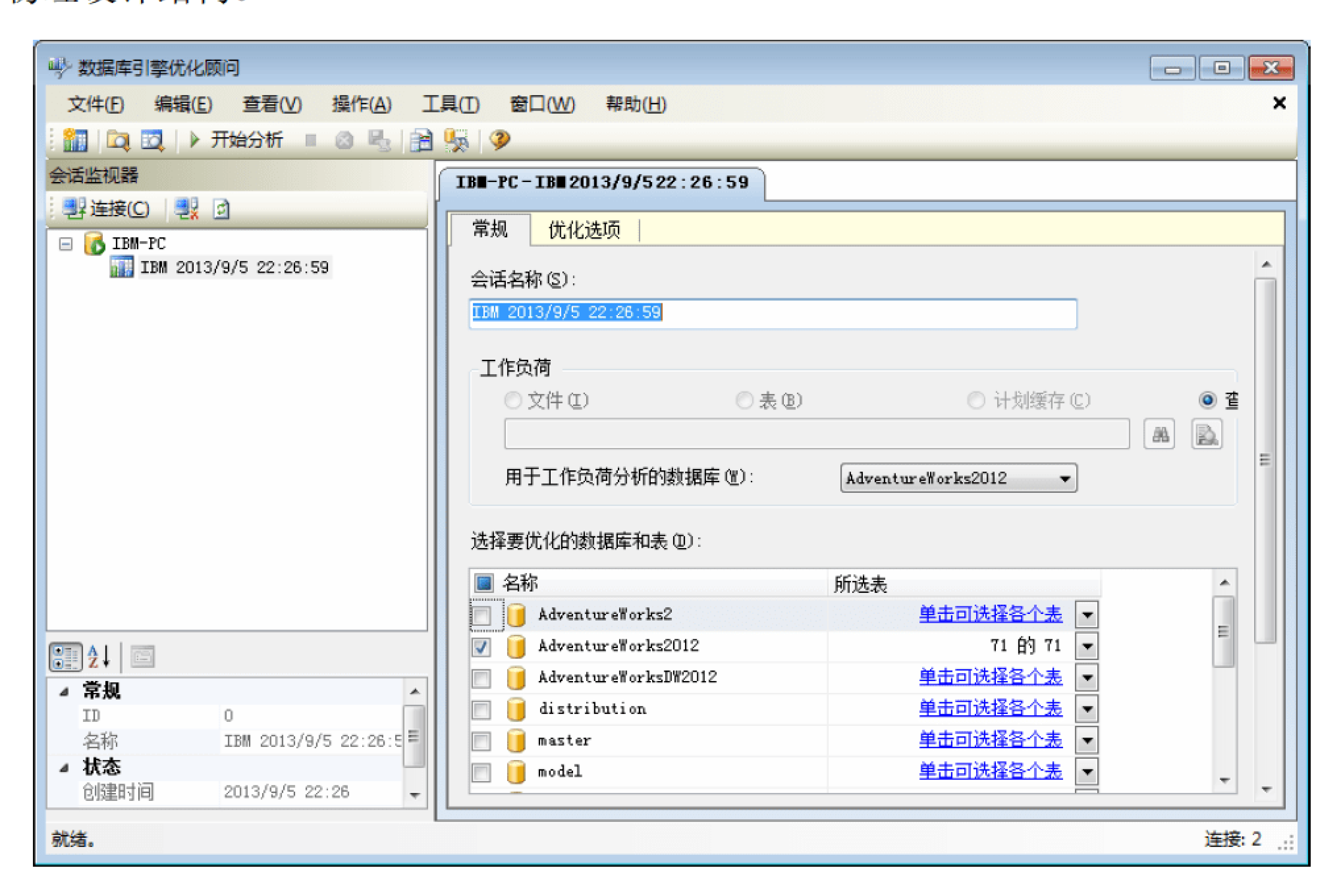


图 22.15　优化顾问对话框

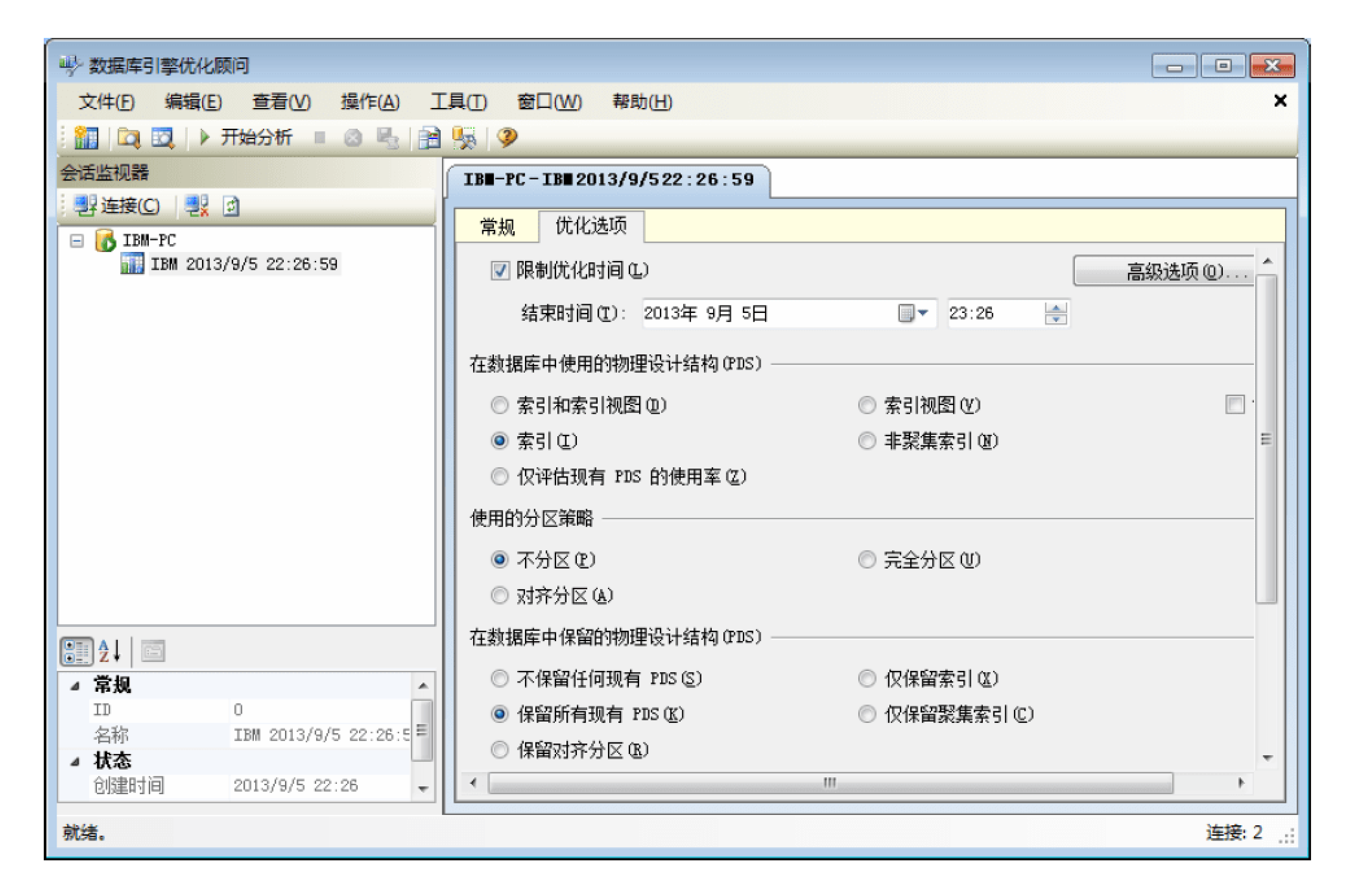


图 22.16　优化选项

（6）单击工具栏中的“开始分析”按钮，优化顾问将根据查询的语句和设置及对应的数据库给出分区建议和索引建议，并给出了应用这些建议后估计效率提高的程度，如图 22.17 所示。

（7）此处给出了索引建议，单击“定义”中的超链接，系统将给出创建该索引的 SQL 脚本预览，定义中的脚本如代码 22.3 所示。可以将该脚本复制到 SSMS 中执行以创建索引，

或者选择“操作”|“应用建议”|“立即应用”命令，将会把索引建议中的修改应用到数据库中。

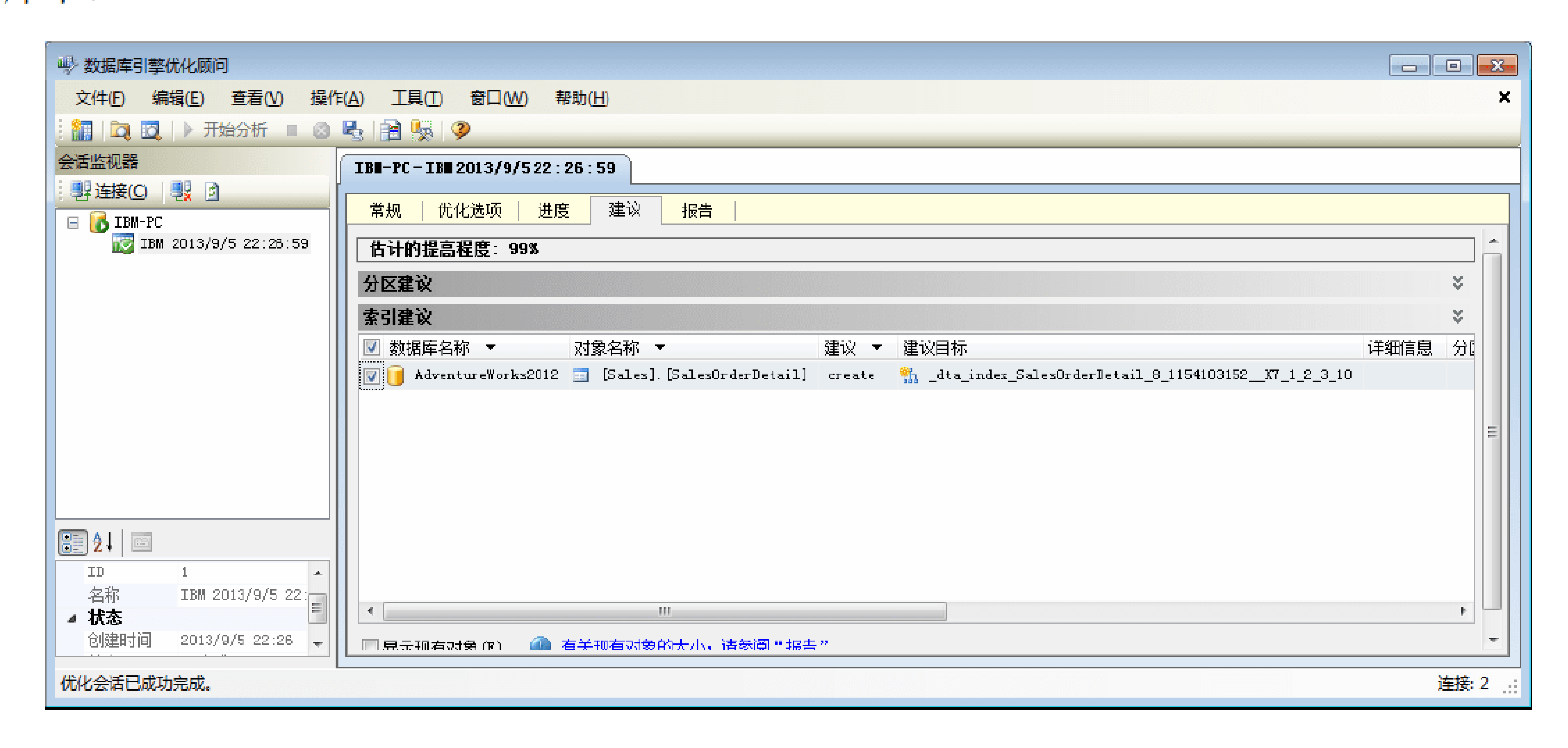


图 22.17　优化建议

代码 22.3　优化建议的脚本

```
CREATE NONCLUSTERED INDEX
[ dta index SalesOrderDetail 14 610101214  K7 1 2 3 10]
ON [Sales].[SalesOrderDetail]
(
    [UnitPrice] ASC
)
INCLUDE ( [SalesOrderID],
[SalesOrderDetailID],
[CarrierTrackingNumber],
[rowguid]) WITH (SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF, DROP_EXISTING
= OFF, ONLINE = OFF) ON [PRIMARY]
```

现在建议中的索引已经应用到了数据库中，可以再次回到 SSMS 执行该查询，在打开了 I/O 统计的情况下可以看到采用建议后 I/O 的变化，读取 6 个数据页，这相对于优化前的 1241 个数据页读取，有了十分明显的改善。

22.5　动态管理视图和函数

SQL Server 从 2005 版开始将核心重新构建，将内部的架构数据和统计数据，通过动态管理视图和动态管理函数呈现给用户。通过查询这些视图和函数，可以观察系统内部当前的执行情况，以监控和调校性能。

22.5.1　动态管理视图和函数简介

动态管理视图（Dynamic Management View，DMV）和动态管理函数（Dynamic

Management Function，DMF）返回可用于监视服务器实例的运行状况、诊断故障及优化性能的服务器状态信息。

DMV 和 DMF 是 SQL Server 2005 增加的新特性，取代了 SQL Server 2000 中的部分系统数据表和函数的功能。DMV 和 DMF 都是在 sys 架构下，而且都使用 dm_开头，通过 SSMS 的对象资源管理器可以查看到所有的 DMV 和 DMF。DMV 位于每一个数据库的系统视图中，一般在通过 DMV 查询系统数据时，需要搭配上 DMF。DMF 的数量较少，集中呈现在 master 系统数据库的“可编程性”、“函数”、“系统函数”、“表值函数”节点之下。

动态管理视图和函数分为服务器实例范围内的动态管理视图和数据库范围内的动态管理视图。由于查询动态管理视图或函数需要对对象具有 SELECT 权限，以及 VIEW SERVER STATE 或 VIEW DATABASE STATE 权限，所以可以通过权限管理有选择地限制用户对动态管理视图和函数的访问。

动态管理视图与一般的视图不同，不需要为每个数据库设置权限，只需要在 master 中创建用户，然后拒绝该用户对不希望被访问的动态管理视图或函数的 SELECT 权限。此后，无论该用户的数据库上下文如何，用户都将无法选择这些动态管理视图或函数。

通过动态管理视图可以查看数据库、执行计划、索引、事务、I/O 等方面的内部信息，如表 22.2 所示列出了动态管理视图的各个类别。

表 22.2　动态管理视图的类别

与变更数据捕获相关的动态管理视图	与查询通知相关的动态管理视图
与公共语言运行时相关的动态管理视图	与复制相关的动态管理视图
与数据库镜像相关的动态管理视图	资源调控器动态管理视图
与数据库相关的动态管理视图	与Service Broker相关的动态管理视图
与执行相关的动态管理视图和函数	SQL Server扩展事件动态管理视图
与全文搜索相关的动态管理视图	与SQL Server操作系统有关的动态管理视图
与索引有关的动态管理视图和函数	与事务相关的动态管理视图和函数
与I/O相关的动态管理视图和函数	与安全相关的动态管理视图
与对象相关的动态管理视图和函数	

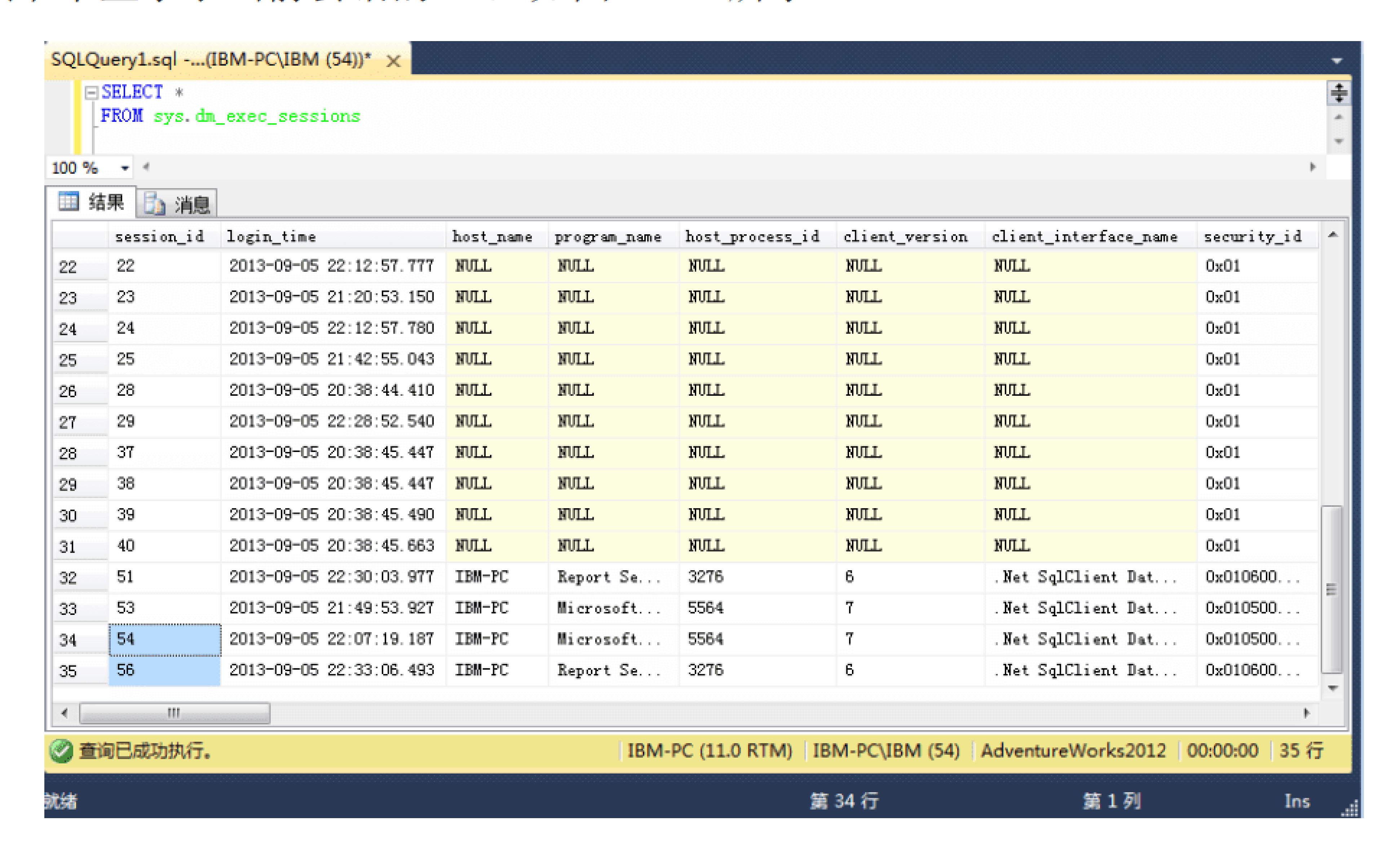
22.5.2　动态管理视图和函数的使用

若要查询当前数据库中的所有会话，则可以查询动态管理视图 sys.dm_exec_sessions，代码如下：

```
SELECT *
FROM sys.dm_exec_sessions
```

在 SSMS 中查询时，若只有当前会话连接到数据库而没有其他应用程序连接到数据库，系统将返回大约 30 行查询结果数据。SQL Server 中规定 session_id 大于 50 的才是外部的会话，而小于 50 的是内部使用的会话。在当前只开启了一个查询窗口的情况下会有 2 个 session_id 大于 50 的会话，一个会话是 SSMS 在对象资源管理器中使用的，另外一个是当前查询的会话。相对于 SQL Server 2005 的 SSMS，SQL Server 2012 的 SSMS 在查询窗口

的选项卡中显示了当前会话的 ID，如图 22.18 所示。



	session_id	login_time	host_name	program_name	host_process_id	client_version	client_interface_name	security_id
22	22	2013-09-05 22:12:57.777	NULL	NULL	NULL	NULL	NULL	0x01
23	23	2013-09-05 21:20:53.150	NULL	NULL	NULL	NULL	NULL	0x01
24	24	2013-09-05 22:12:57.780	NULL	NULL	NULL	NULL	NULL	0x01
25	25	2013-09-05 21:42:55.043	NULL	NULL	NULL	NULL	NULL	0x01
26	28	2013-09-05 20:38:44.410	NULL	NULL	NULL	NULL	NULL	0x01
27	29	2013-09-05 22:28:52.540	NULL	NULL	NULL	NULL	NULL	0x01
28	37	2013-09-05 20:38:45.447	NULL	NULL	NULL	NULL	NULL	0x01
29	38	2013-09-05 20:38:45.447	NULL	NULL	NULL	NULL	NULL	0x01
30	39	2013-09-05 20:38:45.490	NULL	NULL	NULL	NULL	NULL	0x01
31	40	2013-09-05 20:38:45.663	NULL	NULL	NULL	NULL	NULL	0x01
32	51	2013-09-05 22:30:03.977	IBM-PC	Report Se...	3276	6	.Net SqlClient Dat...	0x010600...
33	53	2013-09-05 21:49:53.927	IBM-PC	Microsoft...	5564	7	.Net SqlClient Dat...	0x010500...
34	54	2013-09-05 22:07:19.187	IBM-PC	Microsoft...	5564	7	.Net SqlClient Dat...	0x010500...
35	56	2013-09-05 22:33:06.493	IBM-PC	Report Se...	3276	6	.Net SqlClient Dat...	0x010600...

图 22.18　查询的会话 ID

与 sys.dm_exec_sessions 相似的还有一个 sys.dm_exec_connections，该 DMV 用于显示当前数据库的连接信息，如连接方式、登录认证方式、网络配置等。一般情况下，一个连接对应一个会话，所以查询出的外部会话的个数和连接个数是相等的。

若要查询每一个会话正在执行的情况，可以通过 sys.dm_exec_requests 动态管理视图，该 DMV 显示了当前数据库实例中正在执行的操作时间、状态、命令、执行计划等信息。其中返回的 sql_handle 列是正在执行的 SQL 语句的句柄，通过 sys.dm_exec_sql_text 动态管理函数，可以返回对应的 SQL 脚本文字。由于 sys.dm_exec_sql_text( )是一个表值函数，所以需要使用 CROSS APPLY 语句进行查询。例如查询当前会话执行的情况，对应的查询脚本如代码 22.4 所示。

代码 22.4　查询当前会话执行情况

```
SELECT t.TEXT,r.*
FROM sys.dm_exec_requests r
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) t
WHERE r.session_id=@@spid
```

返回结果的 text 列正好就是当前执行的语句。

如果要查看系统中各个查询的状态，则可以使用 sys.dm_exec_query_stats 动态管理视图，该视图返回查询的创建时间、最后一次执行的时间、执行过的次数、读写情况等信息。在性能调优过程中十分重要。

SQL Server 会将执行过的执行计划缓存起来，在下次执行脚本时直接使用缓存的执行而不需要重新编译 SQL 语句，通过 sys.dm_exec_cached_plans 动态管理视图可以查看整个系统缓存中的所有执行计划信息，包括缓存计划的大小、使用的次数及执行计划的句柄。通过使用 sys.dm_exec_query_plan( )动态管理函数，将计划的句柄传入该函数便可获得具体的执行计划。例如要查询所有缓存的执行计划，则对应的脚本如代码 22.5 所示。

代码 22.5　查询缓存的执行计划

```
SELECT *
FROM sys.dm_exec_cached_plans c
CROSS APPLY sys.dm_exec_query_plan(c.plan_handle) p
```

执行计划将会以 XML 格式返回，单击返回的 XML 链接或者将 XML 内容存为.sqlplan 文件格式就可以看到图形化的执行计划。

另外还有大量的针对 CPU、I/O、事务等对象的动态管理视图，通过这些视图可以获得更多系统的详细信息。

22.6　小　　结

本章主要讲解了在数据库系统的调优过程中常用的几种工具和方法。数据库报表通过与 SSMS 结合，以报表的形式提供整个数据库实例和单个数据的各种信息。SQL Server Profiler 主要用于跟踪 SQL Server 的执行情况，是性能调优过程中必不可少的一个工具。而性能监视器则是 Windows 自带的一款性能跟踪工具，主要用于跟踪 CPU、内存、磁盘和网络的使用情况。数据库引擎优化顾问用于分析工作负荷并提供性能优化建议。通过查询动态管理视图和函数，可以获得大量性能相关的信息，是性能调优过程中的又一件利器。